Declarative CAD Feature Recognition

— an efficient approach

By

Zhibin Niu

SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY AT

CARDIFF UNIVERSITY

SCHOOL OF COMPUTER SCIENCE & INFORMATICS

OCTOBER 2015

To my parents
for their patience and support.

Abstract

Feature recognition aids CAD model simplification in engineering analysis and machining path in manufacturing. In the domain of CAD model simplification, classic feature recognition approaches face two challenges: 1) insufficient performances; 2) engineering features are diverse, and no system can hard-code all possible features in advance.

A declarative approach allows engineers to specify new features without having to design algorithms to find them. However, naive translation of declarations leads to executable algorithms with high time complexity. Inspired by relational database management systems (RDBMS), I suppose that if there exists a way to turn a feature declaration into an SQL query that is further processed by a database engine interfaced to a CAD modeler, the optimizations can be utilized for "free".

Testbeds are built to verify the idea. Initially, I devised a *straightforward* translator to turn feature declarations into queries. Experiments on SQLite show it gives a quasi-quadratic performance for common features. Then it is extended with a new translator and PostgreSQL. In the updated version, I have made a significant breakthrough – my approach is the first to achieve *linear* time performance with respect to model size for common features, and acceptable times for real industrial models. I learn from the testbeds that PostgreSQL uses hash joins reduce the search space enable a fast feature finding.

Besides, I have further improved the performance by: (i) *lazy evaluation*, which can be used to reduce the workload on the CAD modeler, and (ii) *predicate ordering*, which reorders the query plan by taking into account the time needed to compute various geometric operations. Experimental results are presented to validate their benefits.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisors Ralph Martin and Frank Langbein. They provided very excellent guidance. Ralph taught me how to question what I was doing, and how to express ideas clearly. Frank taught me to think about alternatives. I always got timely and patient feedback from them, sometimes even very late at night.

I feel very honored to also have had supervision from Malcolm Sabin. My work is based on his and his previous Ph.D. student Paul James Gibson's work. He provided not just insights about research, but also about life.

I also would like to thank Henry Bucklow and others at ITI TranscenData, who provided CADfix software, CADfix API training, and models. They also provided help with transportation and accommodation when I was visiting.

I wish to express my gratitude to the European Commission and Marie Curie Actions. This work was supported by Framework Programme 7 Initial Training Network Funding under Grant No. 289361. This support enabled my Ph.D. study, as well as co-work with so many excellent partners in the INSIST project and to gain experience at the University of California, Berkeley and Johannes Kepler University. I would like to thank Prof. Vadim Shapiro, Prof. Sara Mcmains, and Prof. Bert Juttler who provided me with many insightful suggestions when I was visiting.

I would like to thank colleagues in Cardiff University Visual Computing group. I am grateful for the advice and discussions with Xianfang Sun, Yukun Lai, Jing Wu and others.

I would like to specially thank my parents for their constant support and encouragement.

Contents

List of Publications

The work described in this thesis has also appeared in the following publications:

- Zhibin Niu, Ralph R. Martin, Frank C. Langbein, and Malcolm A. Sabin.
  Rapidly finding CAD features using database optimization.
  Computer-Aided Design, **69**, 35–50, 2015.
  DOI: 10.1016/j.cad.2015.08.001.

- Zhibin Niu, Ralph R. Martin, Malcolm A. Sabin, Frank C. Langbein, Henry Bucklow.
  Applying Database Optimization Technologies to Feature Recognition in CAD.
  Computer-Aided Design and Applications **12** (3), 373–382, 2015.
  DOI: 10.1080/16864360.2014.981468.

List of Acronyms

AFR automatic feature recognition

AAG attributed face adjacency graph

ASV alternating sum of volumes

ASVP Alternating sum of volumes with partitioning

CAD Computer-aided design

CAM computer-aided manufacturing

CAE computer-aided engineering

CAPP computer-aided process planning

CNC computer numerical control

DB Database

DBP Definition-by-primitives

DBS Definition-by-subfeatures

EAAG extended attributed adjacency graph

FAG face adjacency graph

FEA finite element analysis

MAG	mid-surface adjacency graph
MFAG	manufacturing face adjacency graph
MCSG	minimal condition subgraph
LE	lazy evaluation
PO	predicate ordering
RDBMS	relational database management systems

Chapter 1

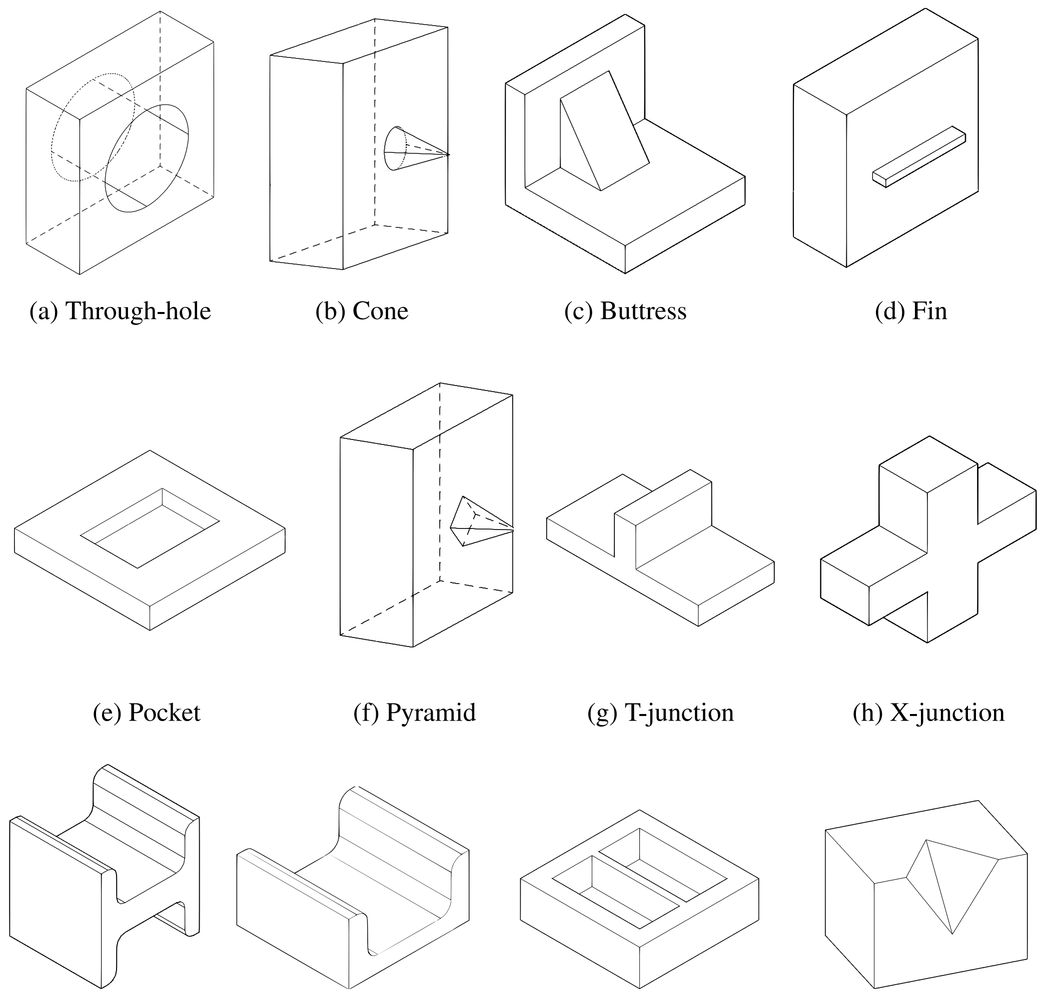
Introduction

1.1 Background

Computer-aided design (CAD) aims to represent real or imaginary objects *truly, completely* and *unambiguously* [TDM$^+$10]. Since the first CAD systems appeared in the mid 1960s, CAD technology has evolved, and is now extensively applied in various field of industry: in design, designers create digital models using geometrical constructs. In mechanical manufacturing, it is integrated with computer-aided manufacturing (CAM), especially in computer-aided process planning (CAPP). Recently, CAD has become much closer to finite element analysis (FEA) in computer-aided engineering (CAE). CAD technology plays an essential role in modern manufacturing industries. It was estimated in 2014 that the overall CAD industry was a $8 billion business [Jon15].

Features in CAD refer to certain substructures associating with certain engineering operations of a solid model. Currently, they are manually identified by engineers using CAD/CAM systems. Features are at an intermediate level between low-level entities (vertices, edges or faces) and high-level entire objects (models). Fig. 1.1, which extends a figure from [LG05], gives some typical (simple) industrial features.

Features play a key, and increasing importantly, role beyond design. Fig. 1.2 gives a typical process flow from model design to manufacturing in industry. In CAD-CAM integration, a typical CAPP system extracts features from a part model and automat-


(a) Through-hole (b) Cone (c) Buttress (d) Fin

(e) Pocket (f) Pyramid (g) T-junction (h) X-junction

(i) I-beam (j) C-beam (k) Rib (l) Notch

Figure 1.1: Common industrial features, including some noted in [LG05]

ically generates computer numerical control (CNC) code. In CAD-CAE integration, models are simplified by removing small features before meshing for analysis. However, many legacy industrial models exist without explicit feature information, or it is absent for other reasons. Even if the model is designed in terms of manufacturing features (Design-by-features in Fig. 1.2) in modern CAD systems [TDM$^+$10], engineering *analysis* features may be different, and engineers may still have to analyse models for different kinds of features than those used for design.

Feature recognition is a reverse engineering task to extract meaningful features from

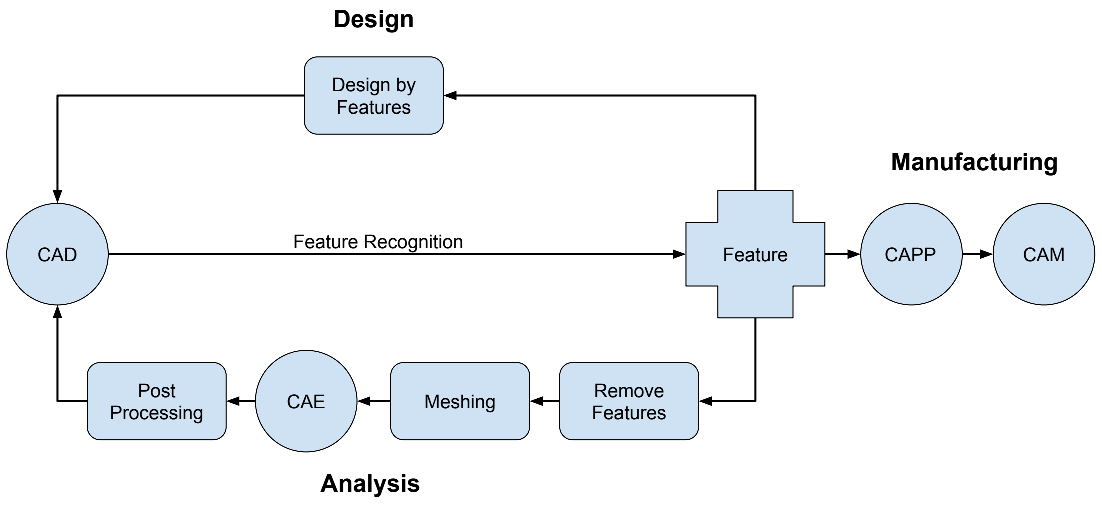


Figure 1.2: Features play a key role in design, analysis and manufacturing

history-free models using computer methodes. Since the seminal work on geometric model analysis and classification by Kyprianou [Kyp80], extensive research has been performed during the past thirty years [ZM02, MNS96, HPR00]. These works are introduced in detail in Chapter 2.

1.2 Research Motivation

Feature engineering has always been driven by real industrial demand. From the start of the 1980s to the end of twentieth century, the major motivation for feature recognition is in CAPP, where feature recognition is used to generate CNC instruction sequences for manufacturing [BNM08]. A lot of work were published and during this stage the main challenge is how to recognize interacted features [HR97, SAKJ01, BNM08].

Recent years, the trend of integrating CAD with CAE has led to a requirement for model simplification [HCB05]. As much as 80% of overall analysis time can be spent on mesh generation in the automotive, aerospace, and shipbuilding industries [HCB05, Cot09]. By removing (typically small) features that have little effect on the analysis results, simplified models can be meshed more quickly and robustly for finite element analysis, and in turn analysed more quickly, as the meshes are simpler [HLGF04, LAPL05, GZL$^+$10, LZM14]. In practice, features are manually found, which is tedious, and in extreme cases, infeasible to carry out reliably, as complex models may have tens of thousands of small features, or more, of many types and forms. Traditional automatic feature recognition (AFR) algorithms face several challenges:

Firstly, features are diverse, and different applications need to find different features. Parts of a shape that are important for machining may be quite different to those which can be ignored during engineering analysis. For example, in manufacturing, a user may want to find through-hole features (see Fig. 1.1 (a)) to generate CNC code. Such features are standard, and can already be handled by commercial software. On the other hand, our partner, the Transcendata company, needed to find notch features (see Fig. 1.1 (l)) during engineering analysis. While such features are also simple, they are rare, and there is no existing software that can reliably find such features in a model. Extra code had to be written to recognize them. In fact, most existing work on feature detection concerns *fixed* algorithms for finding predetermined features [SAKJ01]. However, in practice, it is infeasible to hard-code all possible useful features for all possible domains in advance.

A second issue is that many approaches to feature finding have high computational complexity: times taken to find features can rapidly increase when dealing with complex features and large, detailed models [HR97, SAKJ01].

Thirdly, feature's ambiguity by feature interactions. Interacted features has a different structure/composition comparing to the original features, how to recognize them has always been a tough problem.

The first issue above is challenging as it is difficult for engineering end users to define effective algorithms for finding features. One solution is to use a *declarative* approach: this allows users of a feature finder to simply state what *properties* a feature has, and how a feature is *composed*, rather than having to give a *algorithm* to find instances of the feature.

The performance issue has become a significant bottleneck in industrial CAD-CAE integration, especially as engineering designs are becoming rapidly more complicated. A nuclear submarine may contain 300 times as many parts as an automobile; and for the latter, it can take about four months to prepare a mesh from the CAD model [HCB05]. It is known that classic approaches to finding complex features based on techniques such as subgraph pattern matching, forward chaining using frame-based reasoning, and pattern-matching techniques are high computational complexity [GP92, RGN97]. Recent works on efficient subgraph matching in graph database improved the performance greatly [SWW$^+$12, HLL13]. In graph dabase systems, the subgraph pattern matching problem is either to be solved by breaking the graph into large set of small graphs or to use parallel and distributed computation to accelerate the query on the large graph [SWW$^+$12]. They may provide new theory basis for feature recognition in future. In a declarative approach, naively turning such a definition into an algorithm results in a series of nested loops, which takes far too long to execute for any non-trivial feature. Henderson and Anderson pioneered such a declarative approach [HA84]. Jami Shah and others published a serial of work on declarative approach [SAR94, SBRU95, MSDS04]. Gibson considered six specific optimizations that could be used to transform the naive code into a faster algorithm [GISH97, GIS99]. He used this approach to solve various 2D feature recognition problems. His work will be discussed in more detail in Chapter 2. However, 3D problems involving complex features and large detailed models require further optimization. I sought the required ideas in database query optimization, which is also based on a declarative language.

1.3 Research Hypothesis and Objectives

The hypothesis of this research is that efficient declarative feature recognition can be achieved if the feature declaration is first turned into an SQL query and then processed by the feature recognizer that is built around relational database systems and a CAD modeler.

The Objectives of the research are:

1. To devise a feature language that can describe features declaratively;
2. To devise translation rules by which feature definitions can be turned into SQL queries in a *general* way;
3. To build testbeds to verify the hypothesis, and demonstrate the performance of the feature recognizer;
4. To investigate the execution plan of using database systems to recognize features, this will help to build a standalone feature recognizer without using database systems;
5. To investigate other optimizations beyond database system built-in approaches;

1.4 Thesis Organisation

The rest of this thesis is organized as follows. Chapter 2 discusses previous work. Chapter 3 gives the declarative feature definition language syntax and Chapter 4 overviews the architecture. Chapter 5 presents the SQLite based feature recognizer testbed, experiments on it, and an analysis of how its performance is achieved. Chapter 6 presents the PostgreSQL based feature recognizer testbed, corresponding experiments and performance analysis. Further performance improvements based on lazy evaluation and predicate ordering optimizations are presented in Chapter 7. Conclusions, contributions, limitations and future work are discussed, and a stand-alone feature recognizer is proposed in Chapter 8.

Chapter 2

Related work

2.1 Introduction

Since the seminal work on geometric model analysis and classification by Kyprianou [Kyp80], much work has considered feature recognition. Feature recognition research was relatively more active during the 1980s and 90s, while several reviews were published early in the 21st Century. Following the main classifications used in these reviews [HPR00, SAKJ01, BNM08], the historical work are briefly summarize in this chapter. As noted in Section 1.2, the need for feature recognition is perhaps greater now than ever before. This chapter will first review previous work on feature recognition.

I choose to use declarative approach to recognize features, as does Gibson [GIS99]; I also investigate whether DB query optimization techniques can help to find features more quickly. Thus, in this chapter, the SQL syntax, DB query optimization, and Gibson's optimization as relevant background ideas will be introduced.

2.2 Classic Feature Recognition Approaches

Classic feature recognition systems can be categorized into graph based, volumetric decomposition based, and hint based approaches [HPR00, BNM08]. In this section, I

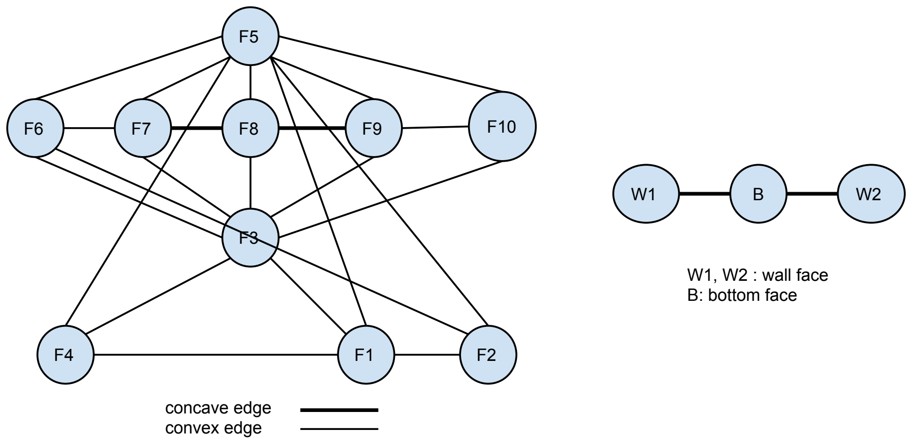


Figure 2.1: Left: graph for model with a slot feature, right: slot template

introduce the main ideas of these classic approaches and examine the performance of various landmark declarative feature recognizers.

2.2.1 Graph-Based Methods

Many successful feature recognition systems are based on graphs [MK90, CC91, FA94, SKG97, LG05, GZL$^+$10]. A fundamental advance was the attributed face adjacency graph (AAG) introduced by Joshi [JC88]. Here, nodes represent faces while arcs denote edges of the solid model. Joshi made an important observation: *for depression features, a face whose incident edges are all convex does not form part of a feature*. He thus decomposes the model graph into subgraphs by deleting all such nodes. The target feature is represented via an AAG; face nodes with all convex incident arcs are deleted. This turns feature recognition into a subgraph isomorphism problem. Fig. 2.1, redrawn from [HPR00], gives an example of how to use a graph template to find features. After removing all convex incident nodes, the model graph only contains nodes {7,8,9}, and is recognized as the same as the template on the right.

Joshi's idea is a graph-based method; it inspired a large amount of work in this field. One main trend was to enrich the expressiveness of the feature graph to solve some issues with the initial approach. Important advances included: Marefat and Kashyap extended the AAG to include geometric constraints on the orientations of faces, and used virtual links to help recognise interacting features [MK90]. However, generating the virtual links is very slow [SAKJ01]. Fields and Anderson extended the AAG to an oriented face adjacency graph (FAG), where edges record exterior/interior information, enabling a faster algorithm [FA94]. Gao and Shah extended AAG to include several geometric attributes of the nodes and arcs of the graph, improving the ability to recognise interacting features [GS98]. Lock et al. performed graph matching on a mid-surface representation, allowing the recognition of features for which the classic approach fails, e.g. because they are composed of complex freeform faces, and not sequentially machined [LG05]. Corney proposed a tailored graph search algorithm for depression/protrusion feature recognition [Cor93].

Graph-based methods can successfully benefit from the well-developed mathematics of graph theory. However, they still have several drawbacks.

Firstly, they are less successful at coping with interacting features and features with variable topologies, such as n-sided bosses for arbitrary n. Several hints may be added to help recognize interacting features; I will discuss this idea later.

Secondly, they are slow. Performance is important real in engineering tasks, especially as real engineering designs become increasingly complex. In general, subgraph isomorphism is an NP-hard problem so typically exhibits worst-case exponential complexity [VR04]. Graph matching based methods have long been criticized for their high computational complexity, and various methods have been proposed to overcome this problem.

Traditionally, a feature template has fixed upper bounds on its size so that the graph can be processed [HPR00]—a typical graph template has at most *six* face nodes [TK94]. Beyond this size, some partitioning strategy or hints may be used [SAKJ01], but even then times can be too long for large models or complex features. Other approaches are also used to improve performance. Field [FA94] defined five classes of machining feature and used oriented face adjacency graph search to achieve linear performance. However, the system only supports prismatic machined features. Regli exploited distributed computing to provide a system with complexity between $O(n^2)$ and $O(n^5)$, depending on the particular configuration of geometric entities and implementation details [RGN97]. Feature vectors can also be used to optimize graph-based matching, achieving $O(n^3)$ performance [VR04]. The approach first turns subgraphs into adjacency matrices, and then groups and orders different elements, finally encoding ordered adjacency matrices as vectors, reducing subgraph isomorphism problems to ones involving three nested loops. Simple declarative approaches also suffer from similar performance problems, as a naive execution plan involves multiple nested for-loops, as previously noted. Recent development of graph database provides new ideas of how to use parallel computing to achieve efficient subgraph matching [SWW$^+$12, HLL13], however such a system are usually rather sophisticated and aims for exploring heavy web-scale graph data. Most recently (10th November 2015), Babai at the University of Chicago announced that a new algorithm efficiently solves the graph isomorphism problem [Bab16]. This provides new theoretical foundation for the classic graph based feature recognition.

Thirdly, it is difficult to extend the approach to real industrial tasks involving complex geometry and topology [RGN97].

2.2.2 Volume Decomposition Methods

Volume decomposition approaches are also quite general, but better at dealing with interacting features [LCCT98]. Such methods usually decompose a CAD model into a set of intermediate volumes which are then classified to produce features [HPR00]. A key idea is the convex hull decomposition approach [Woo82, Kim92, SD96] which aims to generate CNC machine steps for machine features. It usually uses four consecutive steps:

1. Alternating sum of volumes with partitioning (ASVP) decomposition. Firstly, determine a polyhedral convex hull around the part. Then recursively define the alternating sum of volumes (ASV) as the difference in volume between the part and its convex hull. Using a remedial partitioning procedure— ASV with partitioning (ASVP)—for curved shapes, the method converges [Kim92]. Mathematically, for a machined model P, as it is usually machined from its convex hull, a point exists set satisfying:

   $$CH(P) = P -^* CHD(P)$$

   where $CH(P)$ means the convex hull of P, the smallest convex point set that contains the model, $CHD(P)$ means convex hull difference, and the symbol $-^*$ is defined as the regularized set difference. The first step is to generate a tree of convex hulls.

2. Find form features (form features are just shapes, while machining features have functional or manufacturing properties). The ASVP components are classified to form features if they have two or more transitively connected original faces. This step can recognize basic features such slots, ribs, and bosses. However, this method may fail for several special cases [HPR00].

3. Generating primitive machining features.

4. Aggregating machining features.

The ASVP approach finds features by applying different rules to decomposed convex hulls. However, the main issue is that in each step it may generate features that cannot be machined.

Another important approach in volume decomposition methods is cell based decomposition [SC93, SC94]. This approach usually first decomposes delta volumes of a model into cells and then composes them into recognizable features. However, deciding how to combine the large amount of cells into suitable features is not easy, and this has exponential time complexity [HPR00].

2.2.3 Hint-based Methods

Hint-based approaches were proposed to deal with feature interaction problems [HPR00]. Vandenbrande and Requicha [VR93] define a series of minimal unavoidable presence rules based on the hypothesis that any machined feature must leave traces produced by machining operations even for intersecting features [HPR00]. Typical hints include nominal geometries, design features, tolerances, etc. Hint-based approaches are computationally efficient for small features but depend on the generation and definition of hints [FOL+03], and refer to hard-coded features—it is not easy for end users to modify them or define new features [SAKJ01]. Key papers include [VR93, GS98, BDS08]. The most recent important hint based approach is [GS98]. Here, Gao extended the FAG to an extended attributed adjacency graph (EAAG) by adding several attributes. For example, for edges he added convexity, existence, loops, geometry and blend types, while for faces he added source, convex hull, the number of loops, split status, and geometry. The manufacturing face adjacency graph (MFAG) is defined as a connected subgraph of the EAAG of a part, in which no node represents either a stock face or a convex hull face. The minimal condition subgraph (MCSG) is defined as the maximal sub-EAAG of a feature that remains in the EAAG of the part. The MCSGs are used as hints when performing graph matching. They report the time complexity to be $O(M * N * T * (M + N))$ where M is the maximum number of edges per face, T is the maximum number of arcs per MFAG, and N is the maximum number of nodes per MFAG.

Various other approaches have also been suggested for recognizing features, e.g. using octrees to identify assembly features based on spatial and contact face adjacency relationships [SCC01], artificial neural networks to assist in the recognition of complex features [PH92, ÖÖ01, SP09], etc.

2.2.4 Landmark Declarative Feature Recognition Systems

The outstanding key challenges in CAD-CAE integration, as noted, are performance, and the need for end users to be able to define their own features. The latter arises as engineers who understand what a feature *is* may not be expert in devising geometric *algorithms* to find such features. Among the various methods, the declarative approach has the advantage that the end user does not need to be an expert in graph algorithm design. This section will describe research on declarative feature finding and performance achieved.

Martino [DMFG94] developed a *teaching by example* technique for form feature recognition, which first recognizes the protrusions and depressions of the component using syntactic pattern matching then performs graph matching. Suh [SW97] defined features textually in terms of a set of *fundamental features* and their spatial configurations represented by *fundamental spatial relationships*, turning the feature extraction problem into a constraint satisfaction problem. This replaces the usual face adjacency graph search by a hint-based constraint-graph traversal. They also investigated several optimizations to reduce the search space. Performance has worst-case time complexity of $O(mn^2)$ where m is the number of nodes of the relationship graph, and n is the number of fundamental features in the part. The idea to use a declarative approach has been considered previously. N-rep was a declarative system based on EXPRESS [SS91, MSDS04]; it used a graphical interface to allow users to define features by selecting necessary entities. A tailored system for locating turning features in mill-turn parts was developed based on N-rep, with overall $O(n^2)$ complexity where n is the number of machining faces [LS07]. However, its feature recognition performance more generally is unclear. Gibson also considered a declarative approach and six optimizations [GIS99]; I will analyze them in detail in Section 2.6.

2.3 Feature Finding as Data Retrieval

If treating a feature definition as a query and the procedure of feature recognition is applying the query to the data (CAD model), feature finding can be viewed as a structured data retrieval problem. Retrieval must:

1. find *all* features, and must not miss any,
2. find features which *exactly match* the definition.

These requirements necessitate an *exhaustive search* of the CAD model.

Data retrieval can be achieved using various approaches: using Prolog to describe the problem, using relational database systems or graph database systems as the backend to retrieval, or making a stand alone data retrieval system. Prolog– a declarative language, use similar optimizations like relational database systems [wik15e] are used a lot in artificial intelligence problems including feature recognition [HA84], also refer to section 3.6.2 for reasons why I do not choose to use Prolog as the backend of the feature recognizer. Graph database systems are good at dealing with the subgraph matching problems from large graph data, however, they usually requires parallel computing facilities and have various customized query languages for different implementations. I observed that relational DB systems have long been used as the main way of storing and retrieving large amounts of related data. DB systems use a declarative language—SQL—to retrieve information via queries. DB systems typically achieve good performance by automatically choosing low-cost execution plans. I note that SQL queries also require, in principle, *exhaustive search*.

Inspired by relational DB systems, I hoped that if I built a feature recognizer around a DB kernel and a CAD modeler that provides data to the DB, features can be found quickly and the mature DB query optimization technology can be used for 'free'. However, there are several key issues in such an idea, for example, how to turn feature definitions into queries? Can this be made to work for all kinds of features? Will the approach be efficient enough? I designed and implemented testbeds to answer these questions. Later I will give implementation details, but here I first explain some basic concepts of SQL and DB query optimization.

2.4 SQL Syntax

In the approach, feature definitions are translated into SQL queries. As some notation and properties of SQL are used in the translations and explanations, I explain the basic concepts of SQL in this section.

I first explain the syntax of SQL. In the discipline of relational algebra, relational DB systems model data and perform queries using set operators. The actual query language is SQL, which is a high-level declarative language used to implement relational algebra [Mel93]. A typical SQL query is a `SELECT` statement which retrieves data from one or more tables. Listing 2.1 gives a typical SQL query used to retrieve information from a database:

```
SELECT c.tstamp
FROM commits c, actions a
WHERE a.file IN
  (SELECT id FROM files WHERE path = ... )
  AND a.commit_id = c.id
  AND c.id>5
GROUP BY c.tstamp
HAVING agg();
```

Listing 2.1: Example SQL query

A full query is composed of the following clauses:

Target list clause : between `SELECT` and `FROM`. It names the information the user wishes to retrieve from the database. It corresponds to a projection operation in relational algebra.

Range table clause : between `FROM` and `WHERE`. It lists the relations involved in the query by referring to named *range* tables. These tables are the source of the target information. This clause corresponds to a Cartesian product operation in relational algebra. Since computing a Cartesian product (or `CROSS JOIN`) is slow and would often require a prohibitively large amount of memory space to store, the DB query optimizer analyzes the qualification clause and turns it into an `INNER JOIN` if possible. Other set operations allow the user to determine different search spaces for the query, e.g. `OUTER (LEFT, RIGHT, FULL)`, but are not used in this thesis. For more details, see [Cod70]. In SQL, `INNER JOIN` operations are often performed implicitly. For example, in Listing 2.1, the term `a.commit_id=c.id` specifies an `INNER JOIN` query where tables `c` and `a` are linked together via join attributes `commit_id` and `id`. Such a form is the one used in my proposed approach.

Qualification clause : the statement after `WHERE`. It specifies conditions the selected elements should satisfy and corresponds to selection predicates in relational algebra. The predicates may be combined using the logical connectives `AND`, `OR` and `NOT`.

GROUP BY HAVING clause : The `GROUP BY` subclause indicates the columns (must be the same with the target list clause) to group the results and the `HAVING` subclause consists of the conditions under which a group will be included in the final results. The `HAVING` subclause consists of the same conditions that can be used in qualification clause. Both subclauses are optional to the main query, however, the `HAVING` subclause must follow the `GROUP BY` clause in a query.

If the `GROUP BY` subclause is specified, the output is divided into groups of rows and if there is `HAVING` subclause, it eliminates *groups* that do not satisfy the given condition. This reminds us a *hidden* property that is the constraints in `WHERE` clauses are evaluated on all tuples, generating a temporary target list, while the `HAVING` clause further aggregates the temporary (grouped) target list to produce the final results. I will use this idea later.

The target list and range table clauses may sometimes use *aliases*. For example, renaming relations and attributes can be done using `AS` via `old-name AS new-name`.

Qualification and having aggregate clauses mainly fall into one of the following four categories:

Subquery Queries can be nested so that the results of one query can be used in another query via a relational operator or aggregation function. A nested query is also known as a subquery. Listing 2.1 contains the example:
`a.file IN (SELECT id FROM files WHERE path = ... ).`
Different predicates are optimized in different ways. A subquery is a multiblock query (having multiple `SELECT` operations) and is usually turned into `JOIN` queries, as explained later.

Access predicate This is a predicates that links two relations together. For example, the term `a.commit_id=c.id` in Listing 2.1 is an access predicate. Such predicates cause data to be accessed either via index operations or join operations in DB systems.

Index filter predicate This places a constraint on a column which has an index. An example is `c.id>5` in Listing 2.1, assuming there is an index on column `id`.

Table level filter predicate This places a constraint not involving an index, so a full table scan must be performed to exclude irrelevant data. If there were no index on column `id` in Listing 2.1, `c.id>5` would be a table level filter predicate. Every `id` would have to be checked in turn.

Access predicates and filter predicates are different:

1. Access predicates specify starting and stopping conditions for relation traversal, so that only tables meeting conditions in the predicate (requiring an index algorithm) or when matching the columns that join two tables (requiring a join

algorithm) are retrieved. Filter predicates (using indexes of whole tables) consider the column data during a *single* table traversal only. Filter predicates do not contribute to the start and stop conditions, so do not narrow the scanned range [Win15].

2. Access predicates lead to better performance as discard rows *before* they are retrieved from disk or memory, while filter predicates discard rows *after* they are retrieved.

In my testbed, I explored how to translate feature definitions into SQL queries, and I consider the differing performances achieved using subqueries, filter predicates, or access predicates.

2.5 Relational Query Optimization

Last section introduced basic SQL query syntax; and this section will discuss the compiler for the language. I first give an overview, then introduce the performance model and finally, describe some classic query optimization technologies.

2.5.1 Overview

Fig. 2.2 gives a typical relational DB query processing flow. Briefly, the parser checks syntax and verifies existence and correctness of relations, attributes, and others items. The translator turns relations into a relational algebra expression. The optimizer finds a good (if not necessarily the fastest) execution plan and sends it to the evaluation engine.

When the query is executed,the query optimizer is used to determine a suitable plan, or algorithm, from the declarative form of the query. Considerable effort may be put into query planning, as the savings over straightforward plans may be significant, and indeed turn an infeasible query into a feasible one. Query optimization is a mature

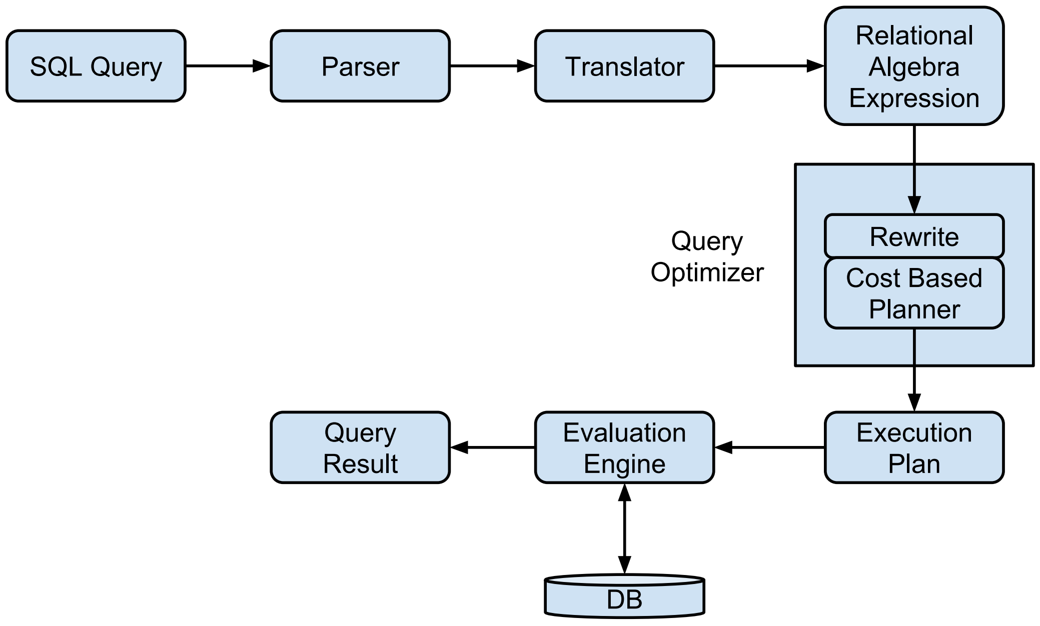


Figure 2.2: Typical RDBMS query processing

field [Ioa96]. Normally, a declarative query is first turned into a relational calculus expression, and the query optimizer then generates various execution paths with equivalent results, using two stages: rewriting, and planning [Ioa96].

The former *rewrites* the declarative query in the expectation that the new form may be more efficient. An example of this approach is *sargable* rewriting (i.e. a transformation to take advantage of an index). In this stage, a multi-block query that includes several select-from-where structures in a single query may be converted to a *single* block query via view merging, nested subquery merging and semijoin-like techniques; for details of these operations see [Cha98].

Planning transforms the query at a procedural level, via relational algebra transformations. A cost (I/O and CPU cost) based planner is used to choose the plan predicted to be fastest based on statistical information about the database. System-R, one of the earliest databases to support SQL, pioneered such optimzation [Cha98]. Its use of dynamic programming to select the best query plan has been adopted by most commercial

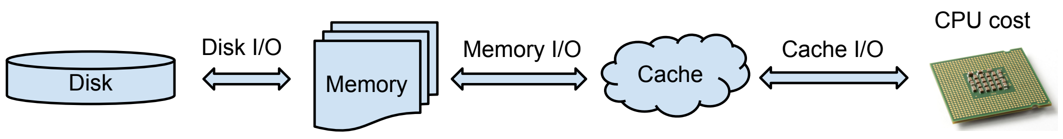


Figure 2.3: Data processing cost model

databases [Ioa96].

For further discussion of query optimization technology see [Mol00].

2.5.2 Performance Model

Data processing performance depends upon I/O and CPU costs. Thus the objective of query optimization is to minimize the cost function:

$$IO\ cost + CPU\ cost.$$

I/O cost refers to the time moving data between different kinds of memory. Fig. 2.3 gives the main costs for typical computers. Specifically they are:

- **Disk I/O**: Moving/writing a block between disk and a main memory *buffer*, perhaps taking 10–30 ms/MB;
- **Memory I/O**: Moving instructions/data between cache and main memory, perhaps taking 10–100 ns/MB;
- **Cache I/O**: Reading/writing data between cache and processor, perhaps taking 10 ns/MB or less;

For a whole data processing procedure, disk I/O dominates other I/O costs. Usually, the number of hard disk block accesses is used to approximate the overall I/O cost. Thus, reducing disk I/O improves performance. A better algorithm requires a smaller number of disk block accesses.

CPU costs in database processing refer to the time spent evaluating `WHERE` statements. These usually involve comparing keys/records, and sorting keys. The overall cost is decided by the number of records, and the cost of evaluating each record. Thus reducing the number of records and the cost of evaluating each record can improve performance. The former requires shrinking the search space, and is performed at the software level, while the latter one usually needs some improvement to the hardware, for example, using solid-state drives to replace the hard disk.

2.5.3 Generalizing Join Sequencing to Reduce CPU Cost

Many queries involve multiple joins. This step finds an efficient execution order in which to process them. Because join tuples are not necessarily symmetric, and the operations are commutative and associative, a translated execution tree with Cartesian products may result in poor performance for some orders of evaluation [Cha98]. One approach is to turn asymmetric one-sided outer joins into equivalent but re-orderable expressions [RGL90] by shuffling `GROUP BY` and `JOIN` [CS95] clauses, an important optimization supported by most current database systems [Hip15, The15a, Bur10, DuB05]. In explaining optimizations of scan methods in `JOIN` processing, I assume that the joins are ordered.

2.5.4 Scan Methods to Reduce I/O

Database systems use various methods, including sequential scans, index scans, and bitmap index scans, to scan tables. Index and bitmap index scanning are much more efficient than sequential scanning because only parts of the table have to be considered [The15a]. The planner chooses an appropriate scan method based on selectivity, a quantity that determines the effectiveness of an index in terms of the proportion of the data filtered out [Mom12].

Sequential Table Scan

A sequential scan reads in each row of the table and checks the columns encountered for the validity of conditions in *sequential (serial) order*. Note that if data are not in memory, the time will also include moving data from hard disk to memory, and it takes longer time than manipulating data in main memory. A sequential scan is I/O cost intensive.

Index Scan

An index is a data structure recording some property of the records. Using an index enables us to quickly find records having that property [Mol00]. Indexes are primarily used to enhance database performance by significantly reducing the number of I/O read operations. Indexing is essentially a kind of lookup table technology. An index must be created prior to executing the main query. After doing so, finding an entry with a certain value does not need to sequentially scan all rows of the table—it can be quickly located by consulting the index table. See [Mol00]. Index scanning often outperforms a table scan, but it requires additional writes and storage space to maintain the index data structure.

The most commonly used index is a B-tree (balanced search tree), which keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. Other index types are supported by various DB implementations. Table 2.1 gives details for several mainstream DB systems (Oracle, MySQL, PostgreSQL, and SQLite). Even different versions of a single DB system may support different index types. For example, MySQL only supported R-tree indexes prior to version 5.5 using the MyISAM storage engine, while later versions support hash indexes. For more details please refer to [wik15b, MyS15a, MyS15b].

Table 2.1 provides basic information which is of use in choosing a DB engine. In my feature recognizer implementations, I first chose to use SQLite as it is lightweight and usually achieves good performance. Later I moved to PostgreSQL for the reason that it

Table 2.1: Index supported by various DB systems

	Oracle	MySQL	PostgreSQL	SQLite
B tree	Y	Y	Y	Y
R tree	Y	Part	Y	Y
Hash	Cluster Tables	Part	Y	N
Expression	Y	N	Y	N
Partial	Y	N	Y	Y
Reverse	Y	N	Y	Y
Bitmap	Y	N	Y	N
Gist	N	N	Y	N
Gin	N	N	Y	N
Full text	Y	Part	Y	Y
Spatial	Y	Part	PostGis	SpatialLite

supports more advanced kinds of indexes and joins. The only index used in this thesis is the B-tree index.

Some DB systems can automatically create an index. For example, when there is no index available to aid the evaluation of a query, SQLite might create an automatic index (which lasts only for the duration of a single SQL statement). Constructing an index takes $O(n\log(n))$ time where n is the number of entries in the table, while a full table scan takes $O(n)$. Thus only when the lookup takes more than $O(\log(n))$ time does SQLite create an index. For example, the query below retrieves all tuples of table `t1` and `t2` satisfying an *access predicate* (`t1.a=t2.c`).

```
SELECT * FROM t1, t2 WHERE t1.a=t2.c;
```

If both tables `t1` and `t2` have approximately n rows, then without any indexes, the algorithm will need nested for-all loops, so takes $O(n^2)$ time. Creating an index on table `t2` takes $O(n\log(n))$ time, the algorithm now uses a `t1` full table scan and a `t2` index scan, giving a retrieval time of $O(n\log(n))$. Together constructing index and performing the query requires overall time $O(n\log(n))$. SQLite determines that constructing an automatic index is the cheaper approach.

Other DB systems do not support automatic indexing and require users to create explicitly an index. For example, in PostgreSQL, the user has to create an index using a command of the form below [Pos15].

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method ]
  ( { column | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [
  NULLS { FIRST | LAST } ] [, ...] )
  [ WITH ( storage_parameter = value [, ... ] ) ]
  [ TABLESPACE tablespace ]
  [ WHERE predicate ]
```

2.5.5 Join Processing to Reduce CPU Cost

A JOIN operation in SQL is translated into a procedural algorithm by the query optimizer. The main algorithms which can be used include a nested loop join, a hash join, or a merge join. Nested loops are normally used for small tables, but the other approaches work much better for large tables [Mom12], and are widely used in mainstream database systems [The15a, Bur10, DuB05].

Suppose there are two relations $R(a, b)$ and $S(b, c)$. The JOIN between them is

$$\begin{aligned} &\sigma_{CL} = (R \times S) \\ &CL = C_1\ AND\ C_2 \ldots OR \ldots NOT \ldots \end{aligned} \tag{2.1}$$

where σ is a SELECT operator with a condition list (CL) which contains conditions connected by Boolean operators. SELECT produces a new relation from the old one. Here, $\times$ denotes the PRODUCT operation, generating a new relation consisting of attributes of R and S. The above expression also has a short form $(R \bowtie S)$ where $\bowtie$ is the join operator.

The query planner can translate such a query in various ways:

Nested Loop Join

A nested loop join is a naive algorithm to process a `JOIN` operation. It joins two sets by using two nested loops.

```
For each r in R do
    For each s in S do
        if r.a = s.b then
            output r,s pair
```

Listing 2.2: Nested Loop Join algorithm

A nested loop join algorithm has $O(n^k)$ CPU cost where k is the number of nested loops. However, if the inner loop is replaced by an index scan, the performance can be improved.

Hash Join

Hash joins were developed in [BE77, DG85]. In SQL, a hash join algorithm is usually used to turn a join operation into an algorithm based on *access predicates*. Thus it only effective for a query involving equality of two relations. The idea is to join two tuples only if their hash values are the same, giving a reduced search space for other constraints' evaluation. A classic hash join algorithm has two steps:

Build stage build a hash table of tuples from the smaller relation R on the joining attribute. For example, the join in Eq. 2.1 has access predicate $(r.a = s.b)$, and the system creates a hash table on column $R.a$.

Probe stage scan relation S sequentially. Compute the hash value for each tuple in S, and use the hash value to probe the hash table of R. If a match is found, output the pair, and if not then drop the tuple from S and continue scanning S.

```
//Build stage:
//hash on join attributes r(a)
For each tuple r in R do
    put tuple r into the hash table, based on the value of hash(r.a);

//Probe stage:
```

```
//hash on join attributes s(b)
for each tuple s in S do
    if value of hash (s.b) is a nonempty bin of hash table for R
//if s hash key matches any r in bucket concatenate r and s
    then output s, {r} pair.
    ( {r} are all tuples whose value of hash (r.a) fall to the same bin with s );
```

Listing 2.3: Hash join algorithm

A full analysis of time complexity is given by [Sha86]. Usually, the complexity is $O(m+n)$ where m is the number of entries in the hash table for R and n is the number of entries in the lookup table for S.

(Sort) Merge Join

Merge join, or sort join, is a typical two-pass algorithm which means data is first read into main memory, processed, written out to disk and then reread from disk to complete the operation. The main idea is that if tables are sorted by the join attribute, the system just need to scan each table only once to find joined tuples. It usually includes two steps: sorting tables and joining them.

Listing 2.4 gives pseudo code code for merge join, where a<−b denotes assignment of the value of b to a and $|R|$ denotes the number of tuples in R, etc.

```
(1) if R and S are not already sorted, sort them on the join attribute;
(2) i <- 1; j <- 1;
    while (i <= |R|) AND (j <= |S|) do
        if R[i].a = S[j].b then outputtuples
        else if R[i].a > S[j].b then j <- j+1
        else if R[i].a < S[j].b then i <- i+1

//outputtuples procedure
    while (R[i].a = S[j].b) AND (i <= |R|) do
        k <- j;
    while (R[i].a = S[k].b) AND (k <= |S|) do
        output R[i], S[k] pair;
        k <- k + 1;
        i <- i + 1;
```

Listing 2.4: Merge join algorithm

A full analysis of time complexity is again given by [Sha86]. The overall time is the sum of times to sort both tables plus time to scan them. Usually the complexity is $O(m \log(m) + n \log(n))$ where m and n are the numbers of entries in tables as before.

As for indexes, various DB implementations have different join support. Table 2.2 gives the mainstream DB system support for join algorithms. From the table, it can be concluded that PostgreSQL supports fewer join algorithms than Oracle while MySQL and SQLite only support join ordering and no other join processing. Thus, for *access predicates*, PostgreSQL and Oracle can use a join algorithm or an index algorithm while SQLite and MySQL can only use an index algorithm.

2.6 Gibson's Declarative Approach and Optimizations

As my work builds on Gibson's, I next describe his contribution in more detail. He suggested that a declarative approach to feature definition could be an effective solution to the problem of allowing user-defined features [GISH97, GIS99, GIS97]. He also noted that naive translation of the declarative form into an execution plan leads to very inefficient algorithms and that optimization is necessary.

He defined features in a language with similarities to EXPRESS [SS91]. Features are based on entities (faces, edges, vertices or subfeatures), and predicates that link them. Such a declaration can be easily rewritten as an algorithm using a set of nested `FOR` loops, one per entity in the definition, and `IF` statements, one per predicate. Executing

Table 2.2: Join algorithms supported by various DBMS

	Join Ordering	Nested Loop	Hash Join	Merge Join	Cluster
Oracle	Y	Y	Y	Y	Y
MySQL	Y	N	N	N	N
PostgreSQL	Y	Y	Y	Y	N
SQLite	Y	N	N	N	N

this takes exponential time in the number of entities in the feature definition, so is infeasible for anything but trivial features. Gibson investigated six strategies for optimizing this basic plan; they are clearly related to those used in database optimization, although Gibson did not consider this point of view. His strategies belong to four categories with respect to their effect on time complexity:

1) Strength reduction and loop re-sequencing Both methods aim to reduce time spent inside each nested loop. For example, suppose A, B and C are each an entity of specific type, and $|A|, |B|$ and $|C|$ are the numbers of that type. Suppose the feature has to satisfy some conditions: condition(a, b) and condition(b, c). The naive nested for-all loop algorithm is shown in Listing 2.5. In this case, the time complexity is $O(n^3)$ assuming $O(|A|) = O(|B|) = O(|C|) = n$.

```
For each a in A do
    for each b in B do
        for each c in C do
            if (condition(a,b)==TRUE)
                if (condition(b,c)==TRUE)
                    output (a,b,c)
```

Listing 2.5: Naive procedure code

Strength reduction moves conditions which have no reference to the loop variable out of loops. Here, this rewrite reduces the number of tests of constraint condition(a, b) from $|A||B||C|$ to $|A||B|$. But the condition(b, c) will still be tested $|A||B||C|$ times.

```
For each a in A do
    for each b in B do
        if (condition(a,b)==TRUE)
             for each c in C do
                if (condition(b,c)==TRUE)
                    output (a,b,c)
```

Listing 2.6: Strength reduction

Loop re-sequencing also reduces the number of tests: it is effective only when combined with strength reduction. For example, when $|A| = |B| > |C|$, the algorithm in Listing 2.6 has same result as Listing 2.7, but in the latter algorithm, condition(b, c) is tested $|C||B|$ times while condition(a, b) is tested $|A||B||C|$ times.

```
For each c in C do
    for each b in B do
        if (condition(b,c)==TRUE)
             for each a in A do
                if (condition(a,b)==TRUE)
                    output (a,b,c)
```

Listing 2.7: Loop re-sequencing

Table 2.3 compares these algorithms. It is clear that when $|C| < |A|$, the loop re-sequencing is an improvement, subject to condition(a, b) and condition(b, c) taking the same CPU cost.

Overall, strength reduction and loop re-sequencing *adjust the search space*. Thus the number of conditions tested is reduced, but the overall time complexity remains unchanged: both optimizations only improve performance by a constant factor.

Strength reduction is also used in DB optimization and is performed in the rewritinge stage (see Fig. 2.2). The rewriter usually performs such a transformation at the declar-

Table 2.3: Test times of various algorithm

	Naive algorithm	Strength reduction	Loop re-sequencing
condition(a, b)	$\|A\| \times \|B\| \times \|C\|$	$\|A\| \times \|B\|$	$\|A\| \times \|B\| \times \|C\|$
condition(b, c)	$\|A\| \times \|B\| \times \|C\|$	$\|A\| \times \|B\| \times \|C\|$	$\|B\| \times \|C\|$

ative level. Loop re-sequencing optimization also exists in DB optimization but is performed during query planning. Because it uses statistics, the query planner usually carries out join ordering optimization as discussed in Section 2.5.3. For more details of this optimization, please refer to [HS93].

2) Entity classification and featuretting These both use the idea of splitting a feature definition into several parts and treating them separately. The result is that the whole search is split into several smaller scale searches, improving performance. The difference between the two algorithms is how to split a declarative definition into parts.

Entity classification removes an entity involved in a condition out of the definition, saving the results as a list (thus entities are classified into useable and useless ones). The corresponding conditions are now implemented as list lookup. For example, starting from the naive nested loop algorithm in Listing 2.8, using entity classification, Listing 2.9 can be obtained.

In this example, `list_b` has a reduced search space, reducing overall algorithm time for testing.

```
For each a in A do
    for each b in B do
        if condition(b) == TRUE do
            if condition (a,b) == TRUE do
                output a,b
```

Listing 2.8: Naive nested loop algorithm

```
For each b in B do
    if condition (b) == TRUE do
        add b to list_b

For each a in A do
    for each b in list_b do
        if condition(a,b) == TRUE do
            output a,b
```

Listing 2.9: Entity classification

Table 2.4: Test times are reduced after entity classification

	Naive algorithm	Entity classification
condition(b)	$\|A\| \times \|B\|$	$\|B\|$
condition(a, b)	$\|A\| \times \|B\|$	$\|A\| \times \|B\|$

Table 2.4 gives the number of tests for both the naive algorithm and the entity classification based algorithm. Again, the overall computational complexity is unchanged. Overall, the entity classification splits the query into several parts, computes the most simple conditions,and caches the results to provide a reduced search space for other constraints.

Featuretting refers to splitting a feature into subfeatures. Unlike entity classification that focuses on extracting the most simple *entity* out of the main loop, featuretting optimization extracts*substructures* out of the main loop. Thus, a whole query can be split into several simpler ones, with reduced complexity. Gibson proposed complicated rules on how to divide the main query in [GIS99]. Here, I only illustrate his ideas using an example:

```
For each a in A do
    for each b in B do
        for each c in C do
            for each d in D do
                for each e in E do
                    if condition (a,c) == TRUE do
                        if condition (b,d) == TRUE do
                            if condition (a,b,e) == TRUE do
                                output a,b,c,d,e
```

Listing 2.10: Naive algorithm

```
For each a in A do
    for each c in C do
        if condition (a,c) == TRUE do
            add a,c to list_ac

For each b in B do
    for each d in D do
        if condition (b,c) == TRUE do
        add b,c to list_bd

For each e in E do
    for each ac in list_ac do
        for each bd in list_bd to
            if condition (a,b,c) == TRUE
             output a,b,c,d,e
```

Listing 2.11: Featuretting optimization

Table 2.5 gives the number of tests before and after featuretting optimization. In the naive algorithm, each condition is evaluated $|A||B||C||D||E|$ times, while after optimization, condition(a, c) and condition(b, d) are evaluated many fewer times. Also, condition(a, b, e) is evaluated many fewer times as in practice the number of entries in $list_ac$ and $list_bd$ will be many fewer than $|A||C|$ and $|B||D|$.

Overall, featuretting optimization reduces the time complexity from $O(n^k)$ to $O(\max(n_1^{k_1}, \ldots, n_m^{k_m}))$ where m is the number of parts and n_i is the number of entities in part i. Database systems do not typically automatically split queries into several simpler queries—queries are usually performed on all tuples of the search space, so the database engine optimizer can not do this. However, if the user defines features in terms of subfeatures (a natural divide-and-conquer approach to problem solving), such a split is achieved manually, reducing time complexity.

Featuretting is similar to entity classification, where a subfeature query and the main query are executed separately. The difference is that featuretting rewrites a feature definition as a root (common) feature and a group of featurettes (subfeatures), and performs searches locally, while entity classification aims to separate the most simple

Table 2.5: Test times are reduced after Featuretting optimization

	Naive algorithm	Featuretting optimization
condition(a, c)	$\|A\| \times \|B\| \times \|C\| \times \|D\| \times \|E\|$	$\|A\| \times \|C\|$
condition(b, d)	$\|A\| \times \|B\| \times \|C\| \times \|D\| \times \|E\|$	$\|B\| \times \|D\|$
condition(a, b, e)	$\|A\| \times \|B\| \times \|C\| \times \|D\| \times \|E\|$	$\|E\| \times (\|A\| \times \|C\|) \times (\|B\| \times \|D\|)$

condition (with only one constraint), put the entity into a lookup table, and perform another query.

3) Indexing Precomputing an index allows relevant entities to be directly retrieved, rather than having to check all entities during query processing. This effective technique is used both in Gibson's approach and database engines. In Section 2.5.4, I already gave a detailed analysis of indexing. Time improvements depend on the selectivity of the index.

4) Assignment This approach narrows the search space by finding `WHERE` statements containing equalities and associated conditions. The key idea is to replace an inner loop by results satisfying outer loop conditions, reducing the time complexity.

```
For each a in A do
    for each b in B do
        if a == condition (b) do
            output a,b
```

Listing 2.12: Naive algorithm

```
For each b in B do
    output condition(b), b
```

Listing 2.13: Assignment optimization

2.7 Gibson's Optimizations and DB Query Optimization

There are similarities in nature between Gibson's declarative feature recognition and DB queries. Both perform exhaustive search for certain structured data, and both lead to nested for-all loops if naively translated. Here, I classify Gibson's optimizations into two categories and compare them with DB query optimizations.

Single query optimization Gibson's approaches to optimize the for-all loop are strength reduction, entity classification, indexing, and assignment. It is clear that strength reduction and loop re-sequencing still lead to nested loops. but with reduced numbers of tests. Both optimizations improve performance by a certain factor. Using the index in a nested loop, the full algorithm uses a full scan for outside-loop-entries, while inside-loop-entries uss table lookup, reducing the order of complexity of the nested loops. Similarly in assignment optimization, a nested loop becomes a full scan and a function call, again reducing the complexity order is reduced.

Query split optimization Gibson's other approaches (featuretting and entity classification) are different ways to split a for-all loop into several parts, which again can reduce the complexity order.

DB systems focus on optimizations for a *single query*, and usually it is the user's responsibility to split a query into several parts if deemed useful or necessary. The main automatic optimizations are join reordering, indexing, and choice of join algorithm.

Join ordering is similar to Gibson's loop resequencing but more powerful: DB reordering of join optimization allows use of an index to reduce the time complexity of nested loops. Gibson's loop resequencing has to be used together with strength reduction and its motivation is to avoid some unnecessary tests in the nested loop. Obviously, the former can achieve better performance.

Indexing is effective in both DB optimization and Gibson's approach. In practice, DB systems support various indexes (see Table 2.1), which may be effective for various types of data; some DB systems can automatically establish indexes based on statistics and choose to use the cheapest one. Again, these are more powerful than Gibson's approach.

The join algorithm is another major difference. DB join optimization turns the naive for-all nested loop into a hash join or merge join. Both have lower time complexity than nested loops. Using such join algorithms, the search space is reduced greatly, while Gibson's single query processing is limited to a nested for-all loop search space.

On the other hand, Gibson's query split optimizations do not exist in DB query optimization. By splitting a complex nested loop into several smaller ones, and each is performed in a smaller search space, the time complexity is also reduced. In this case, his single query optimization can also be applied to the smaller tasks. Obviously, these are useful in practice.

I choose to use a DB query optimizer as the kernel of the feature recognizer as it provides more powerful single query optimizations, which I assume will be useful for a feature recognizer intended to process large 3D models. It is also noted that Gibson's query split optimizations can be used as preprocessing to help achieve better performance. In the testbed, I simply permit users to define a feature in terms of subfeatures.

Chapter 3

Declarative Feature Definition

3.1 Introduction

A programming language, in terms of the theory of computation, is a formally constructed language designed to communicate instructions to a machine [wik15d]. Programming languages may be categorized as imperative and declarative. Imperative programming requires explicit execution steps and describes computation in terms of statements that change a program state, and includes procedural, and object-oriented languages. In contrast, declarative programming expresses the logic of a computation without describing its control flow. It includes functional and logical programming languages. Declarative programming describes what computation should be performed without saying how to compute it, as an algorithm. For more details please refer to [RH04, AU92].

Most classic approaches to feature recognition are imperative approaches, but in practice, engineers may wish to find unusual or application specific features that are not hard-coded in systems. A declarative approach enables them to define as a feature whatever they choose, without needing to consider how to find it. As noted in Chapter 2, some classic works on feature recognition explored describing features declaratively [MSDS04, GIS99]. In such an approach, a feature is defined using *instances* and their *relations* or *attributes*; a feature recognizer can then automatically find instances from the model, without explicit instruction on how to solve the problem.

I choose to use the declarative approach to find features (the original feature definition was provided by Dr.Malcolm Sabin) and in this chapter, I explain the feature recognition language. The language is expanded from Gibson's declarations [GIS99] and EXPRESS [Wik15a]. Like Gibson's language, ours is also incremental and respects a strict *declare-before-use* convention (this is the main difference between Gibson's language and EXPRESS). I enhanced Gibson's definitions in the following ways:

1. I adjusted the syntax slightly: (i) I simplified Gibson's language by removing repeated tokens, for example, `WITH` and `WHERE`, (ii) I changed the language structures to make it simpler for users to understand, and (iii) I replaced tokens to make it more intuitive for user to define features.

2. It supports real data types. 3D model feature recognition encountered in real industrial engineering is challenging because it has more complex spatial structures, and the objectives are diverse. Supporting more data type can enhance the expressive power of the domain specific language, but may bring more complex optimization problems.

3. It supports predicates that may be practical constraints in engineering, while Gibson only uses abstract words such as `links`.

Later, I will show how to generate optimized algorithms for various predicates. I will discuss how to enhance the expressive power in other ways at the end of this chapter.

Assuming that any target feature has a fixed number of entities, thus, the user must be able to define exactly what he/she wants using named entities and relations. For example, a pocket could include three or more vertical faces, but in the approach the user must restrict his definition to a precise number of such faces, as well as other constraints in the definition. Processing features with a variable number of entities is left to future work.

In this chapter, I clarify the data involved, give the syntax of the domain specific language, illustrate operations that describe instance's relations and attributes, and finish with a discussion.

3.2 Data

Data types are given in Table 3.1. I first clarify the two concepts used in feature definition: name and id. When defining a feature, the end-user (i.e. engineer) gives each entity in the definition a *name* so he/she can refer to the entities involved in a feature. Each name has a type, for example, `f1` may represent an instance of the `face` type. These are *formal* names. I defined four type of formal names: `vertex`, `edge`, `face`, `body`; each type consists of an id and some other attributes as specified in Table 3.1. The CAD modeler gives each primitive an id so the user can distinguish each face, vertex, etc,. These are *actual* ids. When the feature recognizer looks for a feature, it matches the formal names to the actual ids when seeking feature instances. Subfeatures are substructures that are composed from vertex, edge and face and can construct to the target feature. Using subfeatures help to define a feature more easily. Subfeature can be any substructures that are defined by the user, for example, it can be a loop or a lump.

Other data types in Table 3.1 are mainly used when specifying constraints. For example, one may require the normal of a face to be (approximately) in a certain direction, in which case the normal is compared to a `vector`, and the difference is measured by an `angle`. The enumerated types of convexitytype, edgetype and facetype are defined according to the CADfix API [ITI15], which is the modeler interfaced with the feature finder. Other CAD modelers may support or require different data.

Table 3.1: Data types in predicates

Data Type	Data
vertex	each vertex has an id and several other attributes (may be directly imported from or computed by the CAD modeler), e.g. coordinates or UV parameters which locate the vertex in the parametric plane.
edge	each edge has an id and several other attributes (may be directly imported from or computed by the CAD modeler). Attribute edgetype has enumerate values: straight, arc, intersection, Nurbs, poly_node, set_track. Attribute convexitytype has enumerate values: convex, concave, mixed, tangential.
face	each face has an id and several other attributes (may be directly imported from or computed by the CAD modeler). Attribute facetype has enumerate values: plane, sphere, cylinder, cone, ellipsoid, torus, Nurbs, blend, frog.
body	each body has an id and several other attributes (may be directly imported from or computed by the CAD modeler), e.g. the volume.
subfeature	a table in database with each column is a primitive (vertex, edge, face or body) name. Subfeatures as a kind of variable data type are generated as intermediate results and the column names are decided by the feature definition such as loops or lumps.
real/int	scalar value defined by user.
point	$[x, y, z]$, a coordinate in 3D space.
vector	$[x, y, z]'$ where x,y,z are scalar values.
box	$[x\_lower, x\_upper, y\_lower, y\_upper, z\_lower, z\_upper]'$
uv_box	$[x\_lower, x\_upper, y\_lower, y\_upper]'$
angle	angle is defined using radians, with domain $(-\pi, \pi)$.

3.3 Syntax

The syntax of the language is illustrated in Listing 3.1. A feature definition is composed of three clauses that are separated by system reserved tokens in uppercase; the rest is written in lowercase.

```
DEFINE <feature> AS
    <entity_type1: name11, name12...>
    <entity_type2: name21, name22...>
    ...
SATISFYING
    <predicate1(name11, name22)>
    <predicate2(name21, value)>
    ...
EXPORT
    <name11 as alias1>
    <name12 as alias2>
    ...
END
```

Listing 3.1: Language structure

System reserved tokens (`DEFINE`, `AS`, `SATISFYING`, `EXPORT`, `END`) demarcate the basic structure of the feature definition. The definition consists of several entities and conditions. The feature, with the name of <feature>, contains the entities listed, of various types, which must satisfy several conditions (predicates).

An entity can be a

Primitive, which refers to the basic entities in the feature, for example, a vertex, edge, or face; each instance has a name and a type as Listing 3.1 shows.

Subfeature, which refers to a substructure of the feature. The approach allows a feature declaration to be built up hierarchically using subfeatures. A subfeature must be defined before using it in a definition. For example, a triangular face is a possible subfeature of the notch in Fig. 3.1.

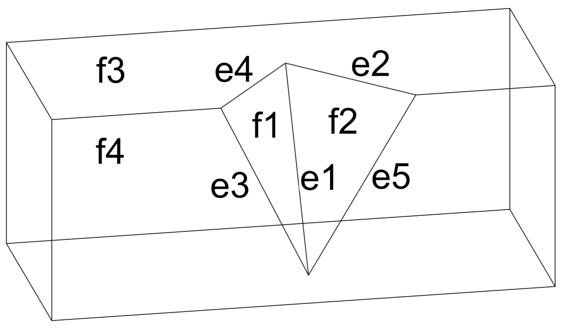


Figure 3.1: Entity naming for a notch.

Different entity types are defined in one line. For each type, the type and instances are separated using a colon. The instances are separated using commas. Each instance in the feature has a name. This name is used as a formal argument in predicates, for example `edge: e1,e2` means `e1` and `e2` are `edgetype` data, and that all edges of the model should be considered as candidates for `e1` and `e2`.

The clauses between `SATISFYING` and `EXPORT` are predicates. Predicates are *atomic operands* of the language. They indicate various constraints that the entities should satisfy. A more comprehensive explanation of predicates will be given in the next section.

The clauses between `EXPORT` and `END` are optional. They are used to define which entities the end-user would like to identify, when this kind of feature is to be used as a subfeature in a further feature definition. See the example in Listing 3.5 later.

3.4 Predicates

Predicates are Boolean-valued functions. Eq. 3.1 gives the basic form of predicates. They may have several arguments (an argument is either a variable or a constant), and return True or False. The variables are symbols capable of taking any constant as value. A predicate is an *atomic operand* in declarative programming languages, and provides much greater power to express ideas formally than propositional variables [AU92].

$$Boolean = Predicate(arg_1, arg_2, ...) \tag{3.1}$$

Predicate logic has proved expressive enough to form the basis of some useful programming languages, such as Prolog and SQL [AU92]. However, as a domain specific declarative language, the *coverage* of the predicates determines the *expressive power* of the language. In practice, it is hard to give a complete set of predicates to describe every attribute a feature may have as the real situation is rather complicated. No classical declarative feature recognition work has fully discussed what an adequate, complete set of predicates might be [Sha91, LS07, GIS99].

Nevertheless, I attempt to predefine some predicates that might be useful in practice. Following the Djinn API [Arm00], I divide the predicates into topological constraints and geometric constraints. Different kinds of predicates require different kinds of translation strategies in the feature recognizer. I note that:

1. The expressive power of any predicate-based declaration is limited by the richness of the CAD modeler's API. Various systems may support different predicates.

2. Ideally, it should be possible for the a programmer (not necessarily the end user) to define additional predicates using the CAD modeler's API.

3. The predicates here involve primitives, as they are more flexible and commonly used in practice than subfeatures. A commercial system could in principle support predicates that understand the structure of a subfeature.

In the next two subsections, I will explain in detail topological and geometric predicates.

3.4.1 Topological Predicates

Its topological structure is a fundamental property of a CAD model. Conventionally, B-rep based modelers provide topological information such as which vertices bound each edge, which edges bound each face, or which edges are incident to each vertex. The topological constraints provide powerful expressions of how the feature is constructed. Important topological predicates are shown in Listing 3.2. In the listing, predicate `Bounds_VE` means vertex v forms one end boundary of the edge e; predicate `Bounds_EF` means edge e forms one boundary of the face f; `Vertex_valency` means there are i edges crossed at vertex v and `Face_valency` means the face is composed of i edges.

```
Bounds_VE(vertex:v, edge:e)
Bounds_EF(edge:e, face:f)

Vertex_valency(edge:f, degree:i)
Face_valency(face:f, degree:i)

Same_id (face: f1, face: f2)
Different_id(face:f1, face:f2)
Greater_id(face:f1, face:f2)
Lower_id(face:f1, face:f2)
// and the same for  vertex_id, edge_id and body_id types

Face_has_number_of_vertices(face:f, int:imin, int:imax)
Face_has_number_of_edges(face:f, int:imin, int:imax)

Body_has_number_of_faces(body:b, int:imin, int:imax)
Body_has_number_of_edges(body:b, int:imin, int:imax)
Body_has_number_of_vertices(body:b, int:imin, int:imax)
```

Listing 3.2: Topology related predicates

3.4.2 Geometric Predicates

Geometric constraints are frequently required in engineering, for instance, a geometric constraint might require some part to have a certain size, or to have a specific type, such as being spherical. Geometric constraints that might be found useful in CAE can be found in [HCB05].

Geometric predicates can be divided into at least the following three categories:

1. **Geometric attribute predicates,** which specify geometric constraints such as the shape of a face or an edge.
2. **Geometric relationship predicates,** which specify some geometric comparison between geometric primitives.
3. **Approximate geometric predicates,** which specifies measurement constraints that are approximately defined, so lead to range-like constraints. For example, engineers may need to find cylinders whose axis is in a certain direction, within a certain tolerance.

The geometric predicates are given in Listing 3.3.

```
//Geometry attributes
    Convexity_is(edge:e1, convexitytype:type)
    Edge_has_geometry(edge:e1, edgetype:type)
    Face_has_geometry(face:f1, facetype:t)

//Geometric relationships
    Edge_longer_than(edge:e1, edge:e2)
    Face_larger_than(face:f1, face:f2)

//Approximate geometry
    Vertex_near_vertex(vertex:v1, vertex:v2, real: rmin, real: rmax)
    Vertex_near_edge(vertex:v1, edge:v2, real: rmin, real: rmax)
    Vertex_near_face(vertex:v1, face:v2, real: rmin, real: rmax)

    Vertex_contained_in_box(vertex:v, box:b)
    Edge_contained_in_box(edge:e, box:b)
    Face_contained_in_box(face: f, box: b)
    Body_contained_in_box(body:b, box:b)

    Face_contained_in_uvbox(face:f, uvbox:uv)

    Face_area_in_range(face:f, real:rmin, real:rmax)
    Edge_length_in_range(edge:e, real:rmin, real:rmax)
    Body_volume_in_range(body:b, real:rmin, real:rmax)

    Sphere_centre_near(face:f, point:p, real:r)
    Torus_centre_near(face:f, point:p, real :r)

    Plane_normal_aligned_within(face:f, vector:v, angle: a)
    Cylinder_axis_aligned_within(face:f, vector:v, angle:a)
    Cone_axis_aligned_within(face:f, vector:v, angle:a)
    Ellipsoid_axis_aligned_within(face:f1, vector:v1, angle:a1,vector:v2, angle:a2)
    Torus_axis_aligned_within(face:f, vector:v, angle:a)

    Sphere_radius_in_range(face:f, real:rmin, real:rmax)
    Cylinder_radius_in_range(face:f, real:rmin, real:rmax)
    Cone_min_radius_in_range(face:f, real:rmin, real:rmax)
    Cone_max_radius_in_range(face:f, real:rmin, real:rmax)
    Torus_radii_in_range(face:f, real:rmin1, real:rmax1, real:rmin2, real:rmax2)
    Ellipsoid_radii_in_range(face:f, real:rmin1, real:rmax1, real:rmin2, real:rmax2,
     real:rmin3, real:rmax3)
```

Listing 3.3: Geometry related predicates (Simple primitive)

A key consideration in defining these predicates is that they are carefully chosen to be simple. This both aids the user who is writing feature definitions, and in translating the definitions into queries. For example, by using `Bounds_EF(edge:e,face:f)`, the user does not need to think in terms of following all edges around the boundary of a face, but simply in terms of *which* edges belong to that boundary.

3.5 Ways to Define Features

Solid models are generally designed hierarchically; they could be built from basic points, lines and so on, but are more typically constructed by combining or modifying simpler models. Similarly, the users are allowed to express a feature in terms of primitives or by using subfeatures.

3.5.1 Definition by Primitives

Definition-by-primitives (DBP) is the most direct way to describe a feature. It requires all related primitives and their constraints (including topological and geometric constraints) to be listed. Primitives in a feature can be divided into:

- **Component primitives:** these make up the feature. For example, in Fig. 3.1, `f1`, `f2`, and `E1` are the main components of the notch feature—they are component primitives of the feature.
- **Support primitives:** these are adjacent to the component primitives. For example, in Fig. 3.1, `f3`, `f4`, `e2`, `e3`, `e4`, and `e5` are adjacent to the component primitives, and are support primitives.

In practice, end-users need to include both component and support primitives into the definition, and specify corresponding topological and geometric constraints. For example, the notch feature in Fig. 3.1 can be defined in this way as in Listing 3.4.

```
DEFINE notch AS
    face: f1,f2,f3,f4
    edge: e1,e2,e3,e4,e5
SATISFYING
//the *_id predicates will be omitted in the new approach in chapter 6.
    Lower_id(f1,f2) //symmetry result breaking, see below
    Lower_id(f3,f4)
    Different_id(e2,e1) //prevent unwanted solutions, see below
    Different_id(e3,e1)
    Different_id(e4,e1)
    Different_id(e5,e1)
    Bounds_EF(e1,f1) // local neighbourhood constraints
    Bounds_EF(e1,f2)
    Bounds_EF(e2,f2)
    Bounds_EF(e2,f3)
    Bounds_EF(e3,f1)
    Bounds_EF(e3,f4)
    Bounds_EF(e4,f1)
    Bounds_EF(e4,f3)
    Bounds_EF(e5,f2)
    Bounds_EF(e5,f4)
    Convexity_is(e1,concave) //boundary edge convexity constraints
    Convexity_is(e2,convex)
    Convexity_is(e3,convex)
    Convexity_is(e4,convex)
    Convexity_is(e5,convex)
END
```

Listing 3.4: Notch feature definition via DBP

In the definition, all edges and faces of the notch feature must be present, as must be the adjacent support faces. Only if all five edges and four faces agree with the various predicates can the feature finder declare the presence of a notch feature. There are four types of predicates used in the notch feature definition:

1. The `Bounds_EF` predicate is a common topological predicate. It rejects edges of the model that do not belong to notch faces.

2. The `Lower_id` predicate is also a common topological predicate. They are used here to prevent the same notch from being found multiple times by symmetry (permuting the labeling of edges and faces would otherwise result in the same notch being found with different identifications for the various edges and faces involved).

3. The `Different_id` predicates prevent unwanted solutions where the same entity is found for things that should obviously be distinct. Without the predicates, results with coincident entities or subfeatures may be found, and in most cases, they are not what the end-user wants. For example, a valid notch does not include two copies of the same triangle face. It is noted that `Lower_id` and `Different_id` predicates are defined manually, and verified that they are necessary by experiments (it can be an interactive feature recognizer if the approach

is efficient enough, so that end users can update his or her feature definitions based on output). The automatic `Lower_id` feature definition are discussed in chapter 8 and the automatic `Different_id` are discussed in chapter 6.

4. The `Convexity_is` predicate is a common geometric predicate, which is an essential character. For example, in the notch finding, the convexity of `e1` determines whether it is a notch or a diamond protrusion.

3.5.2 Definition by Sub-features

Definition-by-subfeatures (DBS) is a more intuitive way to define features where features are constructed from some subfeatures. For example, a notch feature includes two adjacent triangular faces. It can be defined that a 'triangle-face-pair' as a subfeature. Triangular faces are further subfeatures of a triangle-face-pair, as Listing 3.5 shows.

```
DEFINE triangle AS
    face: f
    edge: e1,e2,e3
SATISFYING
    Different_id(e1,e2)
    Different_id(e1,e3)
    Different_id(e2,e3)
    Face_valency(f,3)
    Bounds_EF(e1, f)
    Bounds_EF(e2, f)
    Bounds_EF(e3, f)
EXPORT
    f AS f
    e1 AS e1
    e2 AS e2
    e3 AS e3
END

DEFINE tripair AS
    triangle: t1, t2
SATISFYING
    Same_id(t1.e1,t2.e1)
    Different_id(t1.e2,t2.e2)
    Different_id(t1.e3,t2.e2)
    Convexity_is(t1.e1, concave)
EXPORT
    t1.e1 AS e0
    t1.e2 AS e1
    t1.e3 AS e2
    t2.e2 AS e3
    t2.e3 AS e4
END

DEFINE notch AS
    tripair: n
SATISFYING
    Convexity_is(n.e1, convex)
    Convexity_is(n.e2, convex)
    Convexity_is(n.e3, convex)
    Convexity_is(n.e4, convex)
EXPORT
    n.e0 AS e0
    n.e1 AS e1
    n.e2 AS e2
    n.e3 AS e3
    n.e4 AS e4
END
```

Listing 3.5: Define by subfeature for notch feature

In DBS, when finding a feature, the subfeatures are found and remembered in a table, and the information is propagated back up the hierarchy. Such an approach has similar effects to Gibson's (automatic) featuretting, an optimization technique used to ensure that each search only applies to a local domain. By doing so, later searches only need to consider a smaller set of entities, resulting in lower computational complexity.

In summary, in a declarative approach, the same feature may be defined in more than one way, and without optimization, different definitions may lead to faster or slower ways of returning the same results. If the query optimizer were powerful enough, all definitions would be optimized to the same optimal plan taking the same time. In practice, this does not happen (as I will show later), and so how the engineer writes a definition may have an impact on execution time. A similar problem exists in the declarative programming language Prolog, where programmers must consider procedural aspects of their programs as well as the declarative meanings. A natural way in which the engineer can produce a more efficient definition is to define features in terms of subfeatures. Doing so also helps the engineer break the complex task of feature definition into smaller subtasks. This approach helps to filter the original entities level by level, leaving only feature-relevant data for the next stage of processing.

3.6 Discussion

3.6.1 Expressive Power

As a domain specific declarative programming language, the fundamental capabilities and limitations depend on the completeness of the chosen set of predicates. I give useful predicates that may be useful in practice in Listings 3.2–3.3. Hopefully they will satisfy a large proportion of the requirements of end-users. Although there are many other possible predicates, the set detailed in this Chapter is sufficient to support the experimental investigation of the feasibility.

In real engineering, one may still have difficulty in special cases. A more powerful system might also consider the following:

1. Supporting more predicates by introducing more geometric tools. For example, the medial axis has proven useful in various fields in solid modeling, such as motion planning, shape matching, etc. Some complex features might be expressed most naturally in terms of the medial axis. For example, the thickness of a beam can be obtained quickly in such a way. Ideally, the feature definition language would provide a mechanism for writing new predicates using the CAD system's API.

   This thesis does not cover medial axis because the performance of predicates is limited by the CAD modeler and is task specific. Here, I only focus on the optimizations that are independent of the specific predicates used.

2. Supporting arbitrary functions, not just predicates, would allow expression of ideas in a functional way and thus enhance the expressive power of the language. For example, if a task required the determination of whether the curvature of an edge at a parametric position is in a certain range, it might not be easy for an end-user to provide the desired parameter precisely, but it might be possible for the user to provide a coordinate of some point in space near his target position. A function could then project this point to the nearest point on the edge before calculating the curvature:

   ```
   Curvature_within(get_projection_point(edge_id, point), rmin, rmax);
   ```

   The function (`get_projection_point`) allows the end-user to express his intentions more conveniently.

   In this thesis, I only consider predicates, and not more general functions. Although including the latter would be more powerful, doing so is more a matter of software engineering matters than one that concerns optimization.

3.6.2 The Compiler

The feature recognition language gives end-users a powerful language for defining features. However, I have no intention to implement a brand new programming language from the bottom up; instead, I would like to make best use of existing achievements. I assume that the feature definition language is first turned into an internal representation that is also declarative, and then the latter's compiler could be adapted to solve the task. Various declarative languages exist, and each has its merits. I explain here why choosing SQL instead of Prolog as the internal representation for the test beds.

Prolog, the first predicate logic declarative language, is widely used in education and research. Prolog supports various optimizations to process declarations, including hashing, indexing, tail recursion, etc,. However, it has not had a significant impact on the computer industry [SHCK95], and it is criticised for its poor performance [wik15e].

Another issue is Prolog is not purely declarative: the order of clauses has a significant impact on execution [DEGV01].

SQL, a relational predicate logic declarative language, is by far the most widely used database language. In contrast to Prolog, it is widely used in industry. SQL has various successful implementations including PostgreSQL, MySQL, Oracle, DB2, etc. The SQL compiler turns the declarations into imperative algorithms. A large amount of research has been done on how to choose the cheapest execution plan to give the best performance.

It would be possible to use either SQL or Prolog as the internal representation and corresponding compiler as the basis for this research. Both SQL and Prolog are based on relational algebra, and the query processing ideas are more or less the same. In the testbed, I choose to use SQL query as the internal representation because its success in the industry and the well-developed optimizations that it provides.

Both SQL querying and feature recognition are essentially *exhaustive methods* when expressed at the procedural level. In other words, to achieve the objective—users say what they want, not how to get it—the relational declarative approach has to perform a *complete traversal* of the possible solution space. The ability to reduce the search space and access data of interest quickly is the key issue. Database systems are good at retrieving structured results from a large amount of data, due to the query planner which first generates several execution plans and then chooses the cheapest one.

As noted in Chapter 2, high time complexity has been one of the toughest challenges for classic feature recognition approaches. I now investigate how to turn a feature definition into an SQL query in a general way, so that the DB query optimizer can choose a cheap algorithm for feature finding. I will give a conceptual architecture for this in Chapter 4 and two different solutions in Chapters 5 and 6.

Chapter 4

Feature Recognizer Architecture

I have defined a new domain-specific feature recognition language in the last chapter. A feature recognizer, like a compiler, turns the high-level language into a series of execution instructions. The feature definition is first turned into an internal representation (a SQL query), then a DB engine chooses the cheapest execution plan based on cost estimation. Beyond investigating the power of DB query optimization, I also explored how two other possible optimizations (lazy evaluation and predicate ordering) can help to further improve the performance. This chapter will first give an overview of my feature recognizer architecture, then discusses each module at a high level. Further implementation details of the internal representations will be given in Chapters 5 and 6; lazy evaluation and predicate ordering optimization implementation details will be given in Chapter 7.

4.1 Overview

I now describe the system architecture. Fig. 4.1 shows the conceptual design of the feature finder. The central rectangle encloses all main components of the recognizer. It consists of a translator, a DB query optimizer, and an executor as the main modules. The translator turns the declarative feature definition into an SQL query, the DB query optimizer turns it to execution plan (an imperative algorithm), the executor executes the instructions using either data cached in the database or computed directly by the CAD

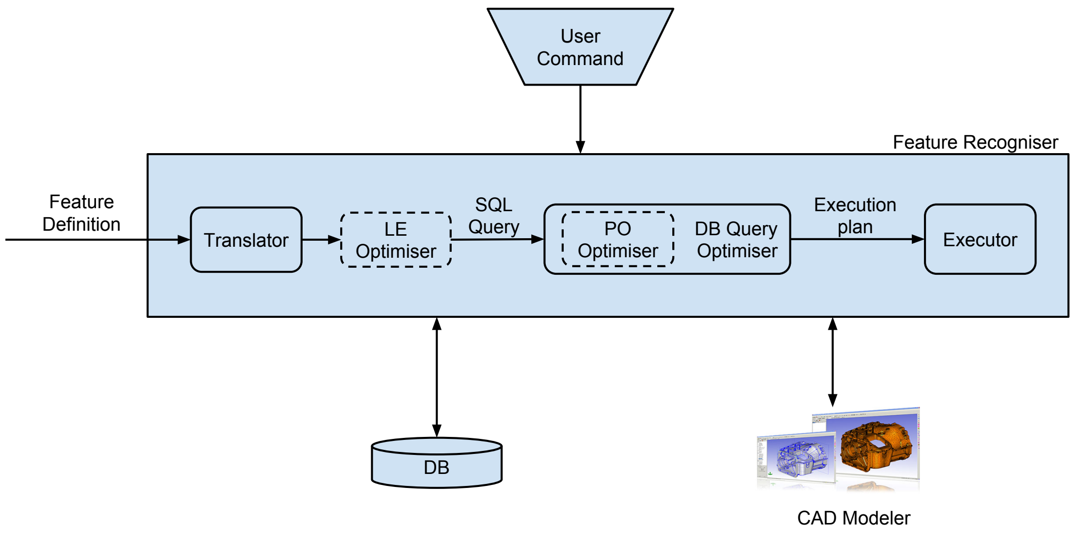


Figure 4.1: Feature recognition architecture. LE: lazy evaluation, PO: predicate ordering.

modeler. In more detail, I have modified the DB query optimizer to support predicate ordering, and developed my own translator and lazy evaluation optimizer to make up the feature recognizer.

Beyond just using best results provided by the DB optimizer, two other optimizations are explored expected to provide further performance improvements: lazy evaluation and predicate ordering. As they were not used in the initial experiments, these modules are drawn in dotted lines. The lazy evaluation is done by SQL rewriting, which happens between the translator and the DB query optimizer. Predicate ordering is deeper and at a lower level—it is performed by the modified DB query optimizer. More details will be given later.

Dynamically, recognizing features including four kinds of interactions:

1. With the end-user: accepting the user's manipulation commands
2. With the definition repository: reading and writing feature definitions
3. With CAD modeler: evaluating and loading data, and highlight the feature on the

CAD model, this allows validation by inspection, refer to example in Fig. 6.5.

4. With DB: caching intermediate data to a local database

These interactions are executed by various modules. For example, the user's manipulation commands are processed by a command analyzer. The translator turns the definition into an SQL query, while the executor takes care of exchanging data between the CAD modeler and DB.

In Figure 4.1, I have omitted some further more detailed modules. They are, firstly, a parser, lexer, and command analyzer. Although they are essential components of the system, they are not relevant to query optimizations; Secondly, I omit a predicate ordering training module, a data import module, etc. Such implementation details will be discussed as needed in Chapter 6.

4.2 Manipulation Language

Following SQL, besides the feature definition language, a complementary feature manipulation language is designed to allow end-users to operate the feature recognizer.

Table 4.1 gives its commands, which are closely related to the four kinds of interaction mentioned above. A command analyzer turns the commands into routines. End-users can ask the feature recognizer to `OPEN` a specified CAD model, `LIST` all existing feature definitions, `DEFINE`, `SAVE`, and `LOAD`, new feature definitions, and to `SHOW` and `PRINT` found features.

A typical work flow involves the user issuing a `FIND` command, assuming a corresponding feature definition has been entered. The definition is translated into an SQL query, and then sent to the database engine to process. The end-user can use the `PRINT` command to output textual details of the features found, or `SHOW` to highlight the features on a drawing of the original 3D model.

OPEN	read in a specified CAD model.
LIST	show existing feature definitions.
DEFINE	input a new feature definition.
SAVE	save a feature definition to disk.
LOAD	read a feature definition from disk.
FIND	find instances of a feature in the model.
SHOW	draw the entities making up a found feature.
PRINT	print details of a found feature.

Table 4.1: Manipulation language

4.3 Main Modules

The essential modules include DB query optimizer, translator and CAD modeler. In this section, I discuss the main considerations of them.

4.3.1 DB Query Optimizer

As noted in the last chapter, the feature definitions are first turned into an internal representation—SQL—that is also declarative, allowing us to adapt its compiler, and the corresponding optimizer and execution engine for the task. Feature recognition is in nature similar to a DB query: both involve *exhaustive searching*. Using SQL as the internal representation and building the testbeds on an existing DB engine instead of writing my own DB query optimizer saved a lot of work, enabling us to focus on optimizations.

The DB query optimizer turns a *declarative* query into an *efficient imperative* algorithm. Initially, I chose SQLite as the DB engine for the feature recognizer for the pragmatic reasons that it is free, open source and has clearly structured code that facilitates linking it to the CAD modeler to build a feature recognizer. SQLite supports a range of query optimization approaches, such as reordering joins, automatic indexing, and subquery flattening. It is easy to revise the code to turn optimizations on and off, to assess their effects.

In my later implementation, PostgreSQL was used as the database engine—it is also free, and open source, which aids understanding of its query optimizer. The changing of the engine has the reasons below:

1. I wanted a *general* translation approach which is applicable to all ((or most) mainstream DB systems; I would like to know whether the translation approach and insights gained from the SQLite testbed were applicable to other DB systems. If not, I would have to reconsider to get a more general translation.

2. I wanted *highly efficient* translated queries that can achieve good performance via the query optimization, and PostgreSQL provides more powerful optimizations than SQLite. There was more chance to obtain better performance using PostgreSQL as the kernel: see Sections 2.5.4 and 2.5.5. The most significant advantages of the query optimizations supported by PostgreSQL over SQLite include:

    (a) Alternative ways to access data using sequential scans, bitmap index scans, or index scans according to filter selectivity (using statistics obtained by `ANALYZE`).

    (b) Alternative ways of processing JOIN operations by shrinking the search space and reducing time complexity, using nested loops, hash joins, and merge joins.

    (c) Reordering join sequences. PostgreSQL's optimizer uses System R's dynamic programming approach when the number of tables is small, but switches to a genetic algorithm to solve the join ordering problem when there is a large number of `FROM` tables [Mom01, GS01].

4.3.2 Translator

The translator turns feature definitions into an internal representation (an SQL query). This is a transformation at the *declarative level*: both are declarations. It also needs to generate efficient SQL, as the DB query optimizer cannot really generate an *optimal* query, but only possibly improve the input query; it may do better at improving some input queries than others. Besides, the translator needs to find a *general* way to finish the declarative rewriting. Specifically it is required that:

1. The translator should be able to process all valid feature definitions.
2. The translated query should be effectively optimisable by the majority of DB systems, not just the ones used in my testbeds.

The first requirement needs a thorough analysis of feature definitions, followed by generating a unified translation framework. The second one urged us to think how to bridge the gap between definitions and queries, while avoiding dependence on any particular DB kernel.

In the initial implementation (the SQLite based feature recognizer), I proposed a *straight-forward* translation approach—the definition is turned an SQL query in which all conditions are expressed as `EXISTS` based subqueries. As SQLite effectively performs indexing optimization, this allows simple features to be found in time approximately $O(n^2)$ for models with n entities, as shown in Chapter 5.

However, on replacing SQLite with PostgreSQL, I found that this approach led to very poor performance. PostgreSQL uses a strategy based on cross-joins via a Cartesian product. Such optimization fails to reduce the complexity of nested loops corresponding to multiple predicates, and even for simple models, could take days to return results. This led us to rethink the way translation was performed. For flexibility, the translator should work in a way that leads to good query processing times independently of the choice of the underlying database engine used. As a result, I developed an inner-join based approach to translation, which is generally applicable to all mainstream DBs, as shown in Chapter 6.

4.3.3 CAD Modeler

The CAD modeler is linked to the feature recognizer, passing model data and results of calculations to the feature recognizer and DB tables. The CAD modeler is responsible for loading shape models, answering geometric and topological queries, etc. The data computed by the CAD modeler are cached but at different stages; I will give details in the testbed chapters.

In my implementations, CADfix [ITI15, BS96] was used as the CAD modeler. It is a commercial geometry translation and repair package primarily intended for 3D model data exchange between different engineering systems and applications. It already provides some defeaturing tools, although I do not make use of these. CADfix (via its API) is used to load CAD models (and repair them to ensure consistent, connected topology), and to interrogate their topology and geometry. It is also used to draw the features found.

4.4 Optional Modules

The basic feature recognizer consists of a translator and DB query optimizer / executor as specified in Fig. 4.1. However, for more sophisticated cases, the end-user may want to find features with certain numerical properties involving angles, areas, distances, etc. in geometric predicates. The required calculations are time-consuming.

It is an obvious fact that it is unnecessary to compute the predicates for all entity instances. Based on this idea, lazy evaluation and predicate ordering optimizations are introduced to gain additional speed. They are regard as optional as they are only useful for computationally-intensive cases.

Lazy evaluation is a strategy that delays the evaluation of an expression until its value is needed, and which also avoids repeated evaluation [Hud89]. In feature recognition, some information is derived data that must be deduced from the CAD model by a computation. Listings 3.2–3.3 give many predicates that require numerical computations, and they may be slow. A better strategy is first to generate a candidate set that satisfies other predicates and then perform such numerical computations only on the candidate set—this will save a lot of unnecessary computing. A general way to turn a feature definition into an SQL query is discovered, but while executing, some constraints are evaluated on all instances while others are evaluated on a candidate set that satisfies other constraints. More details will be given in Chapter 7.

Predicate ordering is a further optimization beyond lazy evaluation. In the testbed, not all data predicates required are pre-loaded: some are computed at runtime when needed (lazy evaluation). However, different predicates may take various times to compute, and the probabilities they return True may differ. For example, in Listing 3.3, `Face_area_in_range` is almost always faster to evaluate than `Face_larger_than` as the second one needs to analyze two faces. Evaluating these two predicates in this given order will typically be faster than the converse, as the former predicate can quickly reject many faces, thus reducing the workload performed by the latter predicate. In practice, not only each predicate's cost but also the probability that this predicate returns True are considered. More details will be given in Chapter 7.

Lazy evaluation and predicate ordering are general optimization approaches and not dependent on any particular DB system. In practice, these ideas are only implemented in the PostgreSQL based feature recognizer. Both approaches are also applicable to SQLite, any other mainstream DB systems, or even a stand-alone feature recognition system.

4.5 Summary

This chapter has considered the architecture of the feature recognizer. The key point is that it turns a declarative feature description into an efficient imperative algorithm (an executor executes the algorithm). The transformation includes several stages: firstly, the feature declaration is turned into an SQL query. If there are some computationally intensive predicates, they are rewritten to ensure lazy evaluation. Secondly, the SQL query is sent to the DB query optimizer where if there are multiple computationally intensive predicates, a predicate ordering optimizer will shuffle the order of predicates based on a metric I discuss later. Thirdly, the DB query optimizer chooses a cheapest execution path (imperative algorithm) based on static analysis of the data. Fourthly, the execution path is turned into machine instructions and executed.

Chapter 5

SQLite Implementation and Quasi-quadratic Performance

5.1 Overview

Having discussed the aims and architecture, I now turn to implementations, and the tests performed upon them. To verify the ideas, a basic feature recognition testbed (only composed of a translator, a DB engine, and an executor) around SQLite and CADfix is built initially. The core question in this chapter is “Can database optimizations help to find features quickly?” with subquestions

1. How should the translator work?
2. What performance is observed in practice?
3. Can the performance are understood by looking at the query plan?

5.2 Testbed Implementation Details

Using the detailed architecture in Fig. 5.1, an SQLite based feature recognizer is implemented. I start by introducing my testbed and then go into translation details. My implementation of Fig. 5.1 is built upon SQLite [SQL15b] by modifying its source code. Here, I give a brief explanation of the workflow.

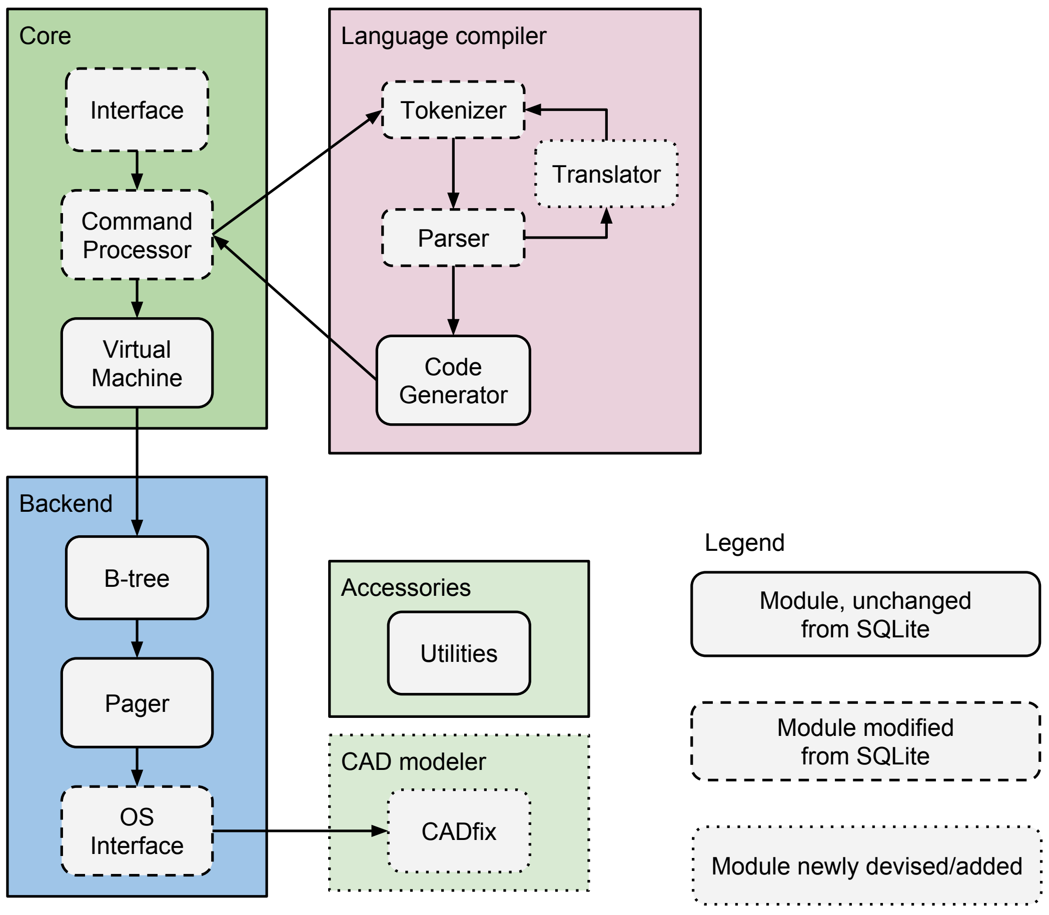


Figure 5.1: SQLite based testbed

The language compiler takes a feature definition in text form, then the tokenizer and parser split it and recast in the form of a parse tree. The translator analyzes the parse tree and turns it into an SQL query that is input to a command processor. The SQL statement is validated by the tokenizer. When the parser reads in the query, it first checks to see if the necessary tables for the entities involved have already been created in the local database (for example, to find instances of a different feature). If not, it issues the necessary SQL `CREATE` commands to generate the tables, and then calls CAD modeler API functions to import the data needed to populate these tables. The CADfix [ITI15] is used as the CAD modeler, as explained in Section 5.3.1. Then the SQL query is again processed by the parser to generate a parser tree.

Query optimizations are performed in the code generator, whose sole job is to convert the parser tree into an algorithm (low level instructions understood only by SQLite) and hand it off to the virtual machine for execution. The accessories module deals with memory allocation and caseless string comparison routines (see [SQL15b] for more detailed explanations of SQLite modules).

As the Figure shows, some SQLite modules are modified, some new modules are added for feature recognition. Specifically, I have:

1. Modified SQLite's tokenizer and parser to support the feature definitions.
2. Added a translator module, which is in charge of turning the feature definition into an SQL query.
3. Modified the interface to the SQLite engine, enabling batch processing for scaling performance experiments.
4. Modified the SQLite backend interface to enable exchanging data with CAD modeler.
5. Modified the command processor, to allow its built-in optimization approaches (re-ordering joins, subquery flattening, and automatic indexing) to be turned off and on, to understand their effects on performance.

The main query optimizations of SQLite come after the command processor module (in the code generator). It has a compact but effective query optimizer [Hip15], which provides sargable rewriting in order to use indexes, and provides algebraic space and method-structure space transformations such as reordering joins, subquery flattening, automatic indexing and group-by optimizations (also using indexing).

5.3 Translation

The translator turns the feature definition into an SQL query. Intuitively, the translation can be straightforward, as both are expressed declaratively. As SQL queries have a close relationship with the data model (range tables), I explain how the geometric data are modeled before describing the translation algorithm.

5.3.1 Data Model

In SQL queries, all relations are expressed as tables. How the data are modeled affects the form of the query expression. The feature definition contains entities and constraints; correspondingly, there are two types of DB tables: entity tables and constraint tables. Entity tables can refer to primitives (faces, edges or vertices) or subfeatures. They are the source relations from which the feature is extracted. The constraints tables (bounds_ef, bounds_ve, convexity, face geometry, etc,.) are the relations used to represent predicates. These models used are as follows:

- primitive entities are modeled as a single column (named as id) table and populated by all instances' id of this type;
- subfeature entities are modeled as a multi-column table where each column is a primitive entity;
- constraints are modeled as a multi-column table where the columns may be a primitive id, and attributes or further primitive ids;

Listing 5.1 gives some examples of these tables. In the entity tables, the `id` is an integer value imported from CAD modeler. Each entity table includes all instances of this type of entity in the model. In the constraint tables, the attributes have data types defined in Table 3.1.

```
//entity tables:
    faces (id int);
    edges (id int);
    vertexes (id int);
    slot(id1 int, id2 int ...);
//constraint tables:
    bounds_ef (edge int, face int);
    convexity(edge int, type int);
    face_has_geometry(face int, geometry int);
```

Listing 5.1: Entity and constraints are modeled as DB tables

There are various ways to populate the tables, including:

Method 1 Extract required data from the CAD model and load it into tables, then just use the tables in normal SQLite queries.

Method 2 Make some virtual tables, but change SQLite so that when it requests data from a virtual table, it instead gets it from CADfix.

Method 3 Do the above, but also cache data as it is retrieved, for efficiency.

In the testbed, for simplicity Method 1 is used as this basic approach let us focus on how the query is optimized when the data are local tables. The updated PostgreSQL based testbed uses either Method 2 or Method 3 to populate tables, as explained in Chapter 7.

In practice, the tables are populated at runtime. As noted in Section 5.2, when the tokenizer reads declarations that require use of some appropriate range table, it determines whether the corresponding table exists. If not, it issues the necessary SQL `CREATE` commands to generate the tables, and then calls CAD modeler API functions to import the data needed to populate these tables. This approach is equivalent to Method 1 in essence, but avoids the need to create tables and populate them manually.

The constraint tables are also loaded using Method 1. The attributes are computed once and all relevant data are cached. For example, if the user asks for the geometry of one face, the geometry of *all* faces is imported to the table.

5.3.2 Translation Rules

The strategy used to translate the feature definition into an SQL query is to map it part by part. The form of a feature delaration is repeated here for convenience, in more detail.

```
DEFINE <feature> AS
    <entity_type1: name11, name12...>
    <entity_type2: name21, name22...>
    ...
SATISFYING
    <predicate1(name11, name22)>
    <predicate2(name21, value)>
    ...
EXPORT
    <name11 as alias1>
    <name12 as alias2>
    ...
END
```

Listing 5.2: Feature definition

A feature definition has the following four parts:

1. `<feature>` between `DEFINE` and `AS` is the target feature name. It is used as the name of the table that stores found instances of the feature. The table's column names are determined from the `<output>` clause.

2. `<input>` is the clause between `AS` and `SATISFYING`. It lists the input entities used to define a feature. These could be primitives (vertices, edges, or faces) or subfeatures.

3. `<constraints>` is the clause between `SATISFYING` and `EXPORT`. These are the basic conditions the feature must satisfy.

4. `<output>` is the clause between `EXPORT` and `END`. It is used to give certain primitives of this feature an alias, and cause information about them to be recorded. It is useful when other features use this feature as a subfeature.

A table named after the feature are generated to cache the found features. Then the `<input>` are translated into a range table clause, `<output>` into a target list clause and the `<constraints>` into a qualification clause (see Section 2.4 for more details). Finally, the three clauses are concatenated to generate the complete SQL query. Pseudocode for this process is shown in Listing 5.3. It is noted that the pseudocode shows the basic idea but does not cover all kinds of predicates (may have more than 3 arguments).

```
1  set root query="CREATE TABLE <feature> AS";
2  set target_list_clause="SELECT";
3  set range_table_clause="FROM";
4  set predicate_clause="WHERE";
5
6  //generate range clause
7  for each input
8      append range_table_clause with '<input> AS <alias_input>';
9  end
10 //generate target list clause
11 for each output
12     append target_list with '<alias_input>.column AS <feature.column>'
13 end
14 //translate predicates
15 for each constraint
16     if (Same_id (v1,v2)) append qualification_clause with v1=v2;
17     // and similar translation for Different_id, Greater_id, Lower_id
18     if (predicate(v1,v2)) append qualification_clause with 'EXISTS ( SELECT <
       constraint>.col1 from <constraint> where <constraint>.col1= v1 and <constraint>.
       col2=v2) AND');
19     if (predicate (v1, rmin, rmax)) append qualification_clause with 'predicate (v1,
       rmin, rmax)) AND')
20 end
21
22 query = append( target_list, range_table_list, predicate_list );
```

Listing 5.3: Pseudocode for translation

In detail, translation includes the following steps:

1. Lines 1–4. Define basic SQL query fragments for the target list, range table, and qualification clauses. The root query creates a table called as `<feature>`. Fur-

ther steps of the translation complete the target list, range table, and qualification clauses separately and combine them with the root query.

2. Generate the range table clause by traversing all `<input>` statements. Range tables contain the source entities the feature is extracted from. As feature recognition requires an exhaustive search, all candidates of each entity type must be considered. Thus, in an SQL query, the same table may be used multiple times, once for each entity of that type mentioned in the feature declaration. They are given different names and, in general, treated differently. Translation of the range table clause gives the entity table an alias name (`entity_name`) for each named entity, for use in the SQL query. For example, `edge: e1, e2;` in the declaration becomes `edges as e1, edges as e2` in the SQL query. The range table can also be a subfeature table, for instance `triangle_face as t1`. The qualifications and target list clause are expressed using the alias names.

3. Generate the target list clause by traversing all `<output>` statements. The target list statements make up the result table (`<feature>`) by specifying each column. As lines 10–12 show, the column of the range table (`<alias_input>`) that user would like to use in other feature definition is given an alias `<feature.column>`.

4. Generate the qualification clause by traversing all constraints. The constraints are translated into condition statements in the qualification clause. The translation is also straightforward. There are three categories of predicates:

   (a) `Same_id(v1,v2)` and similar predicates. These compare the ids of different entities. They are turned into mathematical expressions, for example, `v1=v2`, where `v1, v2` are the column names of the relations. An example using subfeatures is `Same_id(s1.f1,f1)` which is translated into `s1.id1=f1.id` where `s1` is an alias name of a slot subfeature (see Listing 5.1).

(b) `Predicate(v1,v2)`. In such predicates `v2` can be an entity name or a real value. They describe relations between two entities, or state an attribute that the entity `v1` should have. Such predicates are turned into *existence test* subqueries, using `EXISTS` based subqueries. For example, the constraint `Bounds_EF(e1,f1)` is translated into `EXISTS(SELECT bounds_ef .edge from bounds_ef where bounds_ef.edge=e1 and bounds_ef.face=f1)`.

The `<constraint>` table must exist before performing the query (in practice, as noted in Section 5.2, the parser will check for the existence of the table when it reads in a query, and if the relation does not exist, it issues the necessary SQL `CREATE` commands to generate the tables, and then calls CAD modeler API functions to import the data needed to populate these tables.

(c) `Predicate (v1, rmin, rmax)`. Such constraints concern approximate relationshipsin which some property should lie in a target range. They are translated into arbitrary functions that interact with the CAD modeler and return True or False. They are implemented in the SQLite based testbed as in this version I was focusing on optimization of basic feature recognition. They are supported in the PostgreSQL based feature recognition as explained in Chapter 7.

Listing 5.4 and Listing 5.5 shows the results of translation of the notch definition in Listing 3.4 and in Listing 3.5; also see Fig. 3.1.

```
CREATE TABLE notch AS
  SELECT f1.id AS f1, f2.id AS f2, f3.id AS f3, f4.id AS f4,
         e1.id AS e1, e2.id AS e2, e3.id AS e3, e4.id AS e4, e5.id AS e5
  FROM   faces AS f1, faces AS f2, faces AS f3, faces AS f4,
         edges AS e1, edges AS e2, edges AS e3, edges AS e4, edges AS e5
  WHERE  f1.id < f2.id
         AND f3.id < f4.id
         AND e2.id <> e1.id
         AND e3.id <> e1.id
         AND e4.id <> e1.id
         AND e5.id <> e1.id
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f1.id AND bounds_ef.edge = e1.id)
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f2.id AND bounds_ef.edge = e1.id)
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f1.id AND bounds_ef.edge = e2.id)
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f3.id AND bounds_ef.edge = e2.id)
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f1.id AND bounds_ef.edge = e3.id)
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f4.id AND bounds_ef.edge = e3.id)
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f2.id AND bounds_ef.edge = e4.id)
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f3.id AND bounds_ef.edge = e4.id)
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f2.id AND bounds_ef.edge = e5.id)
         AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f4.id AND bounds_ef.edge = e5.id);
         AND EXISTS (SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 1 AND convexity.edge = e1.id)
         AND EXISTS (SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 2 AND convexity.edge = e2.id)
         AND EXISTS (SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 2 AND convexity.edge = e3.id)
         AND EXISTS (SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 2 AND convexity.edge = e4.id)
         AND EXISTS (SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 2 AND convexity.edge = e5.id)
```

Listing 5.4: Notch definition (DBP) as SQL

```
CREATE TABLE triangle AS
 SELECT f.id as f, e1.id as e1,e2.id as e2, e3.id as e3
 FROM   faces AS f, edges AS e1, edges AS e2, edges AS e3
 WHERE  e1.id <> e2.id
        AND e1.id <> e3.id
        AND e2.id <> e3.id
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE bounds_ef.face = f.id AND bounds_ef.edge = e1.id )
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE bounds_ef.face = f.id AND bounds_ef.edge = e2.id )
```

```
       AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f.id AND bounds_ef.edge = e3.id )
       AND EXISTS ( SELECT valency.face FROM valency
                    WHERE valency.degree = 3 AND valency.face = f.id )

CREATE TABLE tripair AS
 SELECT  t1.e1 AS e0, t1.e2 AS e1, t1.e3 AS e2, t2.e2 AS e3, t2.e3 AS e4
 FROM    triangle AS t1, triangle AS t2
 WHERE   t1.e1 = t2.e1
         AND t1.e2 <> t2.e2
         AND t1.e3 <> t2.e2
         AND EXISTS ( SELECT convexity.type FROM convexity
                      WHERE convexity.type = 1 AND convexity.edge = t1.e1 )

CREATE TABLE notch AS
 SELECT  n.e0 AS e0, n.e1 AS e1, n.e2 AS e2, n.e3 AS e3, n.e4 AS e4
 FROM    tripair AS n
 WHERE       EXISTS ( SELECT convexity.type FROM convexity
                      WHERE convexity.type = 2 AND convexity.edge = n.e1 )
         AND EXISTS ( SELECT convexity.type FROM convexity
                      WHERE convexity.type = 2 AND convexity.edge = n.e2 )
         AND EXISTS ( SELECT convexity.type FROM convexity
                      WHERE convexity.type = 2 AND convexity.edge = n.e3 )
         AND EXISTS ( SELECT convexity.type FROM convexity
                      WHERE convexity.type = 2 AND convexity.edge = n.e4 )
```

Listing 5.5: Notch definition (DBS) as SQL

5.4 Experiments

I now describe various experiments carried out to determine if an approach to feature finding based on database optimization is viable, and in particular whether the automatic query optimizer in SQLite can enable features to be found at a reasonable speed. In particular, I consider three questions. Does database optimization help, and if so how much? How powerful is SQLite database optimization? Is this relevant to real models?

5.4.1 Performance Measurements

I clarify performance measurements for feature recognition in this section. Feature recognition includes various operations, and time is spent on at least:

1. computing by the CAD modeler and loading data from it,
2. the DB executing `ANALYZE` to generate statistics for tables (a necessary step before executing a query, as the query optimizer uses the statistics to choose cheapest execution plan),
3. the query optimizer generating possible execution paths;,
4. the query optimizer generating costs of the paths,
5. the query optimizer choosing the cheapest execution plan,
6. the executor generating machine instructions for the execution plan,
7. executing the plan.

In practice, some time costs such as `ANALYZE` and query planning time are more or less fixed. Other costs are beyond my control, e.g. how long it takes the modeler to load the model. Indeed, it is neither necessary nor possible to collect all these times. Instead, I simply consider the feature recognizer as:

1. generating an imperative algorithm, then
2. executing the imperative algorithm.

The first stage turns declarations into algorithms, and it is usually fast. The first step can generate various algorithms, and they may have different performance. I am more concerned with the second stage of performance of the imperative algorithm, as this will become more significant for larger models. The imperative algorithm execution

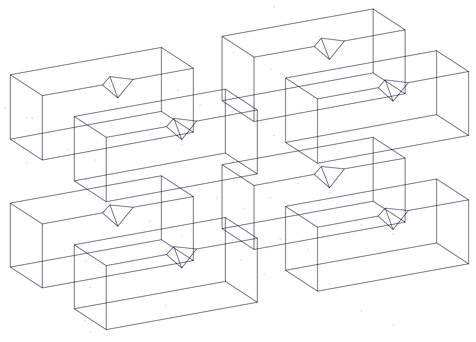


Figure 5.2: Model with 8 notched blocks

time affects the end-user experience directly. Here, I assume that all DB tables can be imported quickly, and ignore the time taken to execute `ANALYZE` or CAD modeler computation time. Thus, I only consider features that are described by topological predicates (Table 3.2), and geometric predicates (Table 3.3). More complicated cases (when one has to consider CAD modeler computation time) will be discussed in Chapter 7.

Two aspects of feature finding performance are considered: absolute times and scaling performance (time complexity). Absolute time gives a baseline that an end-user may experience. However, the absolute time may be dependent on the specific hardware and software. Scaling performance is more important when deciding whether the algorithm is useful or not.

The scaling performance are obtained by experiments on special designed artificial models, and later, by theoretical analysis of execution plans. The performance shown for finding basic features (such as notches, slots and through holes) is discussed in this chapter. Such features are the most common ones engineers need to find in industry [BNM08, SAKJ01]. I explain my artificial models here, as the scaling experiments in this chapter, and the next chapter, were all performed on such models.

Table 5.1: Test platform configuration

Model	Thinkpad W530
CPU	Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
Memory	8GB
OS	Debian GNU/Linux 6.0.10 (squeeze)
Compiler	GCC 4.8.2
SQLite version	3.7.16.2

The artificial models were generated recursively: initially, a model is generated with only one block having only one feature, as shown in Fig. 3.1. Then it is repeatedly duplicated to generate new models with 2, 4, 8, etc. blocks, each with one feature. In this way, a series of models whose edges and faces, and number of features, increasing geometrically are obtained. Fig. 5.2 shows an example after duplicating the model three times. A sequence of such models is named as a *feature model family* in this thesis.

In experiments, it can be estimated the scaling performance using these artificial models. A log-log plot is chosen, in which the vertical t-axis is time taken to find the features of the given type in each model, and the horizontal n-axis is the total number of edges in that model. Assuming that the time taken for feature finding is dominated by $t = \alpha n^p$, fiting a straight line to the log data, its slope gives the exponent p indicating the time complexity.

The test platform configuration is given in Table 5.1

5.4.2 Benefits of Database Optimization

The first experiment aimed to see whether feature recognition can gain benefits from DB query optimization, and if so, how it is achieved.

As already noted, a declarative approach, if naive translated using nested loops, has time complexity $O(n^k)$ for a model with n entities, and a feature composed of k entities (for simplicity ignoring the fact that entities have different types). Clearly, for large models, and any realistic value of k, this is infeasible, and the question is whether the database optimization techniques can significantly improve upon this.

The feature finder was tested on a notch model family (see Fig. 5.2). More experiments on slot and step-rib model family please refer to Fig. 6.4 and Table 6.4. In this section, I focus on using notch family to illustrate the performance and optimizations. The single notch definition in Listing 3.4 has nine entities and seventeen predicates; clearly, naive translation has high complexity. In practice, it represent a typical requirement in engineering analysis: it is simple but rare, and no existing (as far as known) CAD modeler has hard-coded such features; however, engineers may want to remove them before meshing.

The source code of the SQLite query optimizer (the code generator in Fig. 5.1) was modified to enable SQLite's optimizations (reordering joins, indexing and subquery flattening) to be turned on and off. All are disabled but one optimization off to find the benefits of each, and performed an experiment on the feature family. Fig. 5.3 gives the performances for the notch family. It is clear that when all optimizations are turned off, the feature recognizer gives the worst performance, while turning all optimizations on is best, as hoped.

Firstly, the absolute time. With full optimization, the program can analyze a model with 18000 edges in about 5 minutes, which is a realistic, acceptable value for a real feature-finding application. However, in the same time, it can only analyze a model with 1100 edges if the only optimization used is reordering joins, dropping to a model of 70 edges if no optimization is used. Subquery flattening shows little effect in this experiment and the curve when it is the only optimization used is little better than the unoptimized result. Theoretical analysis shows that although the query consists of subqueries, subquery flattening optimization has not proved effective.

Secondly, the scaling performance. Table 5.2 gives slopes for the notch feature and

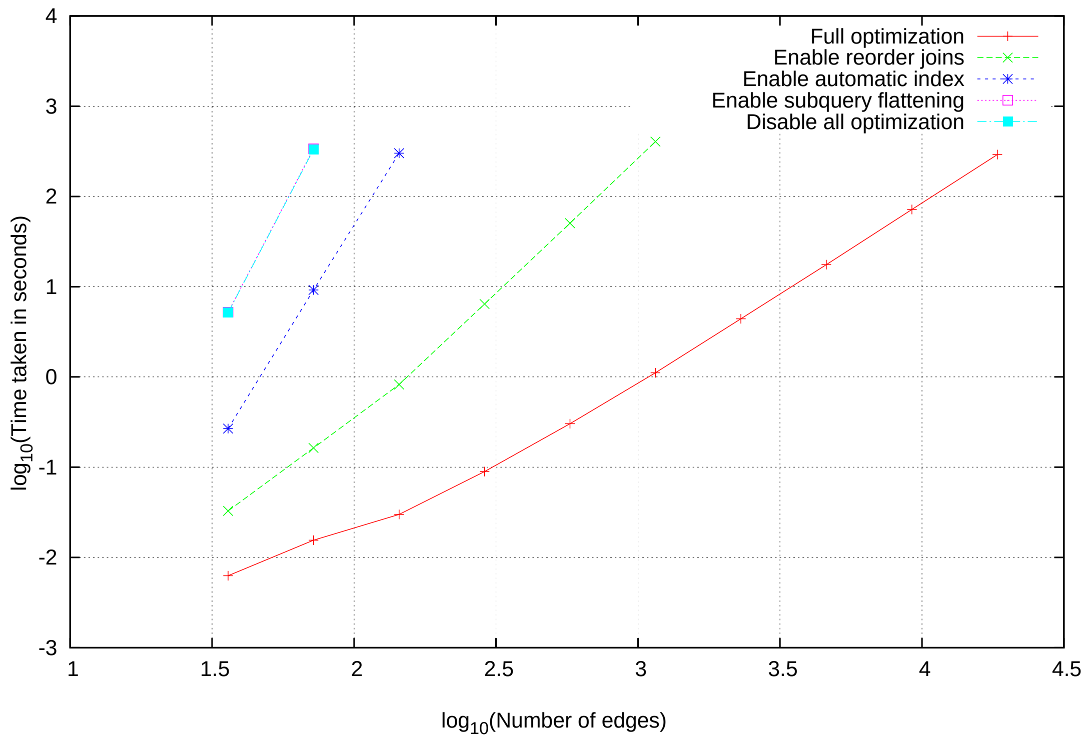


Figure 5.3: Query optimization performance compare (notch)

through-hole (see Listing 5.6) feature using different optimization approaches. It is clear that the fully optimized program has (approximately) a time complexity $O(n^2)$ for the notch feature, while the un-optimized one has a much higher complexity, about $O(n^6)$. Reordering joins by itself helps significantly, reducing the slope to about 3, while automatic indexing by itself has less impact, reducing the slope to about 5. It can be loosely said that, for notch features, that getting 1 order of complexity of benefit from indexing and 3 orders from reordering joins. For through-hole features (defined in Listing 5.6 which is translated as Listing 5.7), the query optimization provided less benefit, due to the somewhat different definition used, reducing complexity from about $O(n^{3.7})$ to $O(n^{2.3})$.

The estimated slopes allow us to predict how the time taken varies as model size increases. For the unoptimized algorithm (indexing, reordering joins and subquery flattening optimizations disabled), a model with a single notch takes about 5 seconds to process, while a model with 8 notches (Fig. 5.2) would take 5×8^6 seconds $\approx$ 2 weeks,

Optimization	Notch	Throughhole
All	2.0	2.3
Reorder joins	2.8	3.4
Automatic indexing	5.0	2.6
Subquery flattening	6.0	3.5
None	6.0	3.7

Table 5.2: Slopes for various optimizations (notch and throughhole)

confirming my assertion that a declarative approach without optimization is infeasible. On the other hand, with all optimizations turned on, the 8 notches model in Fig. 5.2 takes about 32 ms. The much larger model (with about 18000 edges) can be analyzed in a feasible time (about 5 minutes) by the full optimized program while the unoptimized program did not return result after hours. Reordering joins is the most powerful optimization, and automatic indexing is also useful, but subquery flattening shows little effect.

```
DEFINE throughhole AS
face: f1,f2,f3,f4
edge: e1,e2,e3,e4,e5,e6
SATISFYING
Greater_id(f1,f2)
Greater_id(e2,e1)
Greater_id(f4,f3)
Bounds_EF(e1,f1)
Bounds_EF(e1,f3)
Bounds_EF(e2,f1)
Bounds_EF(e2,f4)
Bounds_EF(e3,f2)
Bounds_EF(e3,f3)
Bounds_EF(e4,f2)
Bounds_EF(e4,f4)
Bounds_EF(e5,f3)
Bounds_EF(e5,f4)
Bounds_EF(e6,f4)
Bounds_EF(e6,f3)
Convexity(e1,convex)
Convexity(e2,convex)
Convexity(e3,convex)
Convexity(e4,convex)
Convexity(e5,tangential)
Convexity(e6,tangential)
Face_has_geometry(f3,cylinder)
Face_has_geometry(f4,cylinder)
EXPORT
e1 as e1, e2 as e2, e3 as e3, e4 as e4, e5 as e5,
f1 as f1, f2 as f2, f3 as f3, f4 as f4
END
```

Listing 5.6: Through-hole definition

```
CREATE TABLE throughhole AS
 SELECT f1.id AS f1, f2.id AS f2, f3.id AS f3, f4.id AS f4,
        e1.id AS e1, e2.id AS e2, e3.id AS e3, e4.id AS e4, e5.id AS e5, e6.id AS e6
 FROM   faces AS f1, faces AS f2, faces AS f3, faces AS f4, edges AS e1,
        edges AS e2, edges AS e3, edges AS e4, edges AS e5, edges AS e6
 WHERE  f2.id < f1.id
        AND e1.id < e2.id
        AND f3.id < f4.id
        AND EXISTS ( SELECT face FROM face_has_geometry
                     WHERE  face = f3.id AND geometry = 2006)
        AND EXISTS ( SELECT face FROM face_has_geometry
                     WHERE  face = f4.id AND geometry = 2006)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f1.id AND bounds_ef.edge = e1.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f3.id AND bounds_ef.edge = e1.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f1.id AND bounds_ef.edge = e2.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f4.id AND bounds_ef.edge = e2.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f2.id AND bounds_ef.edge = e3.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f3.id AND bounds_ef.edge = e3.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f2.id AND bounds_ef.edge = e4.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f4.id AND bounds_ef.edge = e4.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f3.id AND bounds_ef.edge = e5.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f4.id AND bounds_ef.edge = e5.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f3.id AND bounds_ef.edge = e6.id)
        AND EXISTS ( SELECT bounds_ef.edge FROM bounds_ef
                     WHERE  bounds_ef.face = f4.id AND bounds_ef.edge = e6.id)
        AND EXISTS ( SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 2 AND convexity.edge = e1.id)
        AND EXISTS ( SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 2 AND convexity.edge = e2.id)
        AND EXISTS ( SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 2 AND convexity.edge = e3.id)
        AND EXISTS ( SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 2 AND convexity.edge = e4.id)
        AND EXISTS ( SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 3 AND convexity.edge = e5.id)
        AND EXISTS ( SELECT convexity.edge FROM convexity
                     WHERE  convexity.type = 3 AND convexity.edge = e6.id)
```

Listing 5.7: Throughhole definition as SQL

5.4.3 Definitions Affect Performance

A declarative approach enables end-user to define features as they like. In Chapter 3, two ways are given to define features: define by entity and define by subfeature. Different ways to define features may induce different SQL query translation and they may lead to different performance, as optimization is not likely to be perfect. In this section, I show how defining features by subfeature affects the performance.

A second experiment is conducted to determine to what extent the input feature definition affects the final optimized performance. Again the notch family models are used. Query optimization was turned on or off for both definitions discussed earlier (see Listings 3.4 and 3.5). Fig. 5.4 and Table 5.3 give the results of this test.

Clearly, the original all-in-one definition was highly inefficient, and query optimization has significantly improved it. However, a subfeature definition based approach is better still. The number of entities in a feature determines the level of loop nesting. The original notch definition is made up of 9 of entities, so the all-in-one definition if

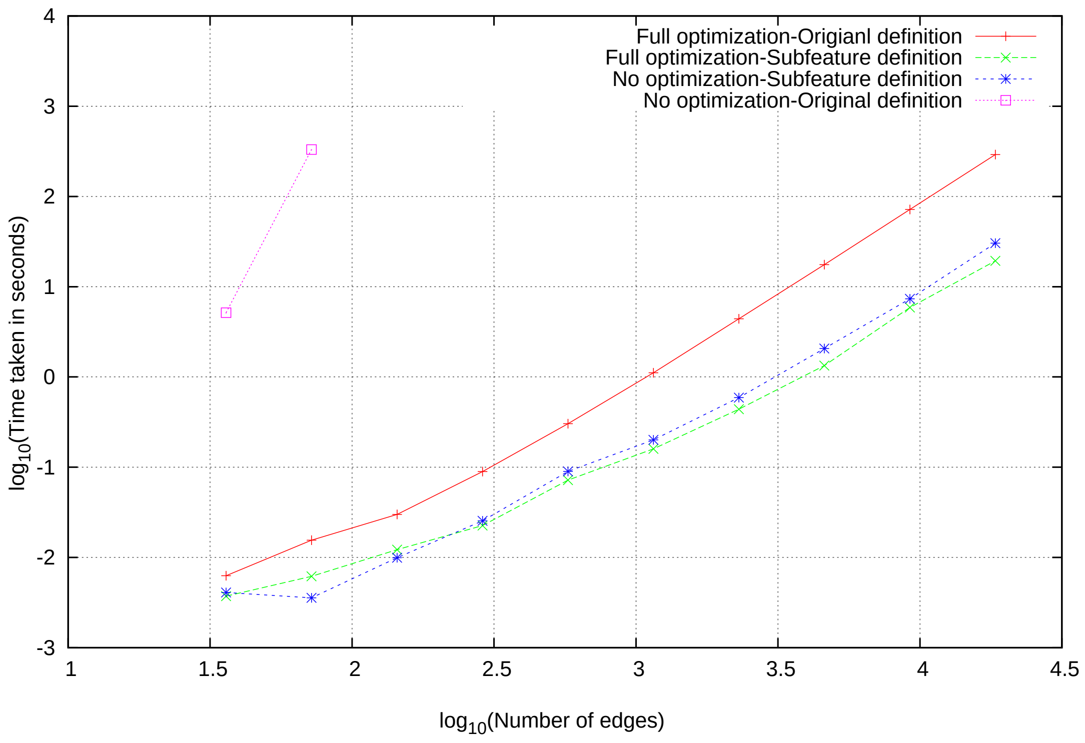


Figure 5.4: Using alternative notch feature definitions

Optimization	Definition	Slope
None	All-in-one	6.0
Full	All-in-one	2.0
None	Subfeatures	1.81
Full	Subfeatures	1.76

Table 5.3: Slopes for various definitions

executed naively would have time complexity $O(n^9)$; in practice, the results are somewhat lower. The subfeature approach has naive time complexity of $O(n^6)$, and again see better in practice, even with optimization turned off.

Nevertheless, it can be seen that, as expected, the subfeature definition is more efficient than the all-in-one definition. Secondly, for the subfeature approach, query optimization has still improved upon the input query, but much less than for the all-in-one approach, as it was more efficient to start off with.

From this experiment, two further conclusions can be drawn. Firstly, although database optimization can speed up feature finding, it does not turn the input query into an *optimal* query: even after database optimization was used, the two definitions took different times to find features. (This is certainly true for SQLite; other database engines may be better at optimization).

Secondly, this in turn implies that the engineer must construct his feature definition carefully. While database optimization can turn a poor definition into a much better execution plan, and can also slightly improve even a good plan, careful thought when constructing the feature definition is also beneficial.

5.4.4 Real Industrial Models

The earlier experiments only considered artificial models, while real industrial models may be very complex and cause feature finders to behave differently. For example, in the previous tests, the number of features went up with model size, but this may or may not happen in real models—there may just be a few features of a given kind on a very complex model.

To explore how the feature finder system performs on at least some simple real industrial models, it is used to find slot features (defined in Listing 5.8, and translated as in Listing 5.9) on a CPU heat sink, a carbine, and a switch, as shown in Fig. 5.5. All slot feature are found correctly, results are summarized in Table 5.4.

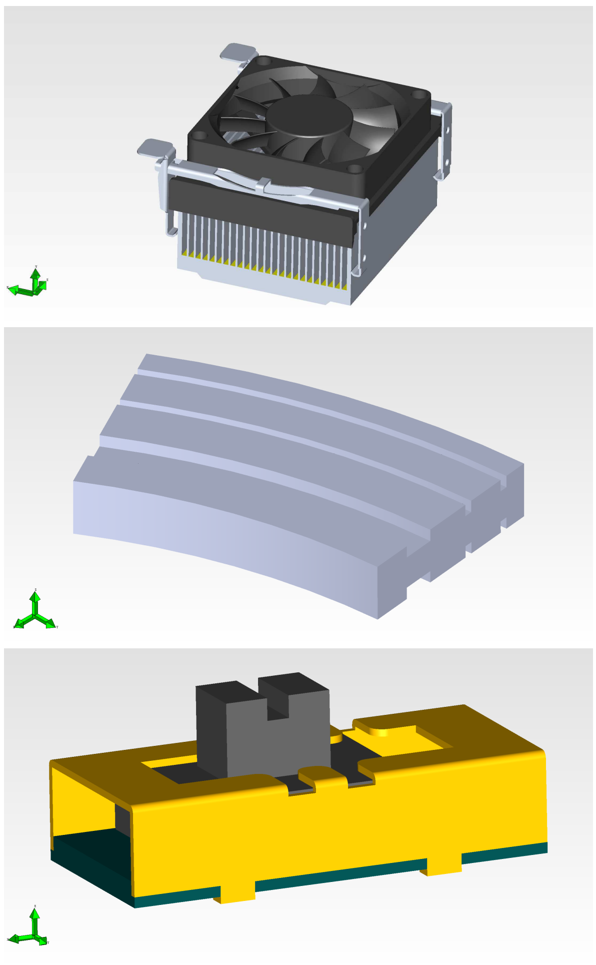


Figure 5.5: CPU heatsink, Carbine and Switch

Absolute time is considered here. For the simplest model, the carbine, optimized feature finding took only 0.05 s; without optimization the time taken was over 14 hours. The more complex switch and CPU heat sink models required 0.2 s and 7 s respectively, and would have taken too long to process without optimization. Using database optimization in combination with a declarative approach to feature definition is thus potentially applicable to real world problems.

```
DEFINE slot AS
face: f1,f2,f3,f4,f5
edge: e1,e2,e3,e4,e5,e6,e7,e8
SATISFYING
Greater_id(e2,e1)
Greater_id(e4,e3)
```

Model	CPU heatsink	Carbine	Switch
Number of edges	2388	84	330
Number of slots	24	6	9
Optimized query	6.94 s	0.05 s	0.22 s
Unoptimized query	—	14.47 hours	—

Table 5.4: Time taken to find slots in real models

```
Different_id(f1,f5)
Different_id(f1,f4)
Face_valency(f1,4)
Bounds_EF(e1,f1)
Bounds_EF(e1,f2)
Bounds_EF(e2,f1)
Bounds_EF(e3,f1)
Bounds_EF(e4,f1)
Bounds_EF(e3,f3)
Bounds_EF(e4,f2)
Bounds_EF(e5,f2)
Bounds_EF(e8,f2)
Bounds_EF(e2,f3)
Bounds_EF(e6,f3)
Bounds_EF(e7,f3)
Bounds_EF(e3,f4)
Bounds_EF(e5,f4)
Bounds_EF(e6,f4)
Bounds_EF(e7,f5)
Bounds_EF(e4,f5)
Bounds_EF(e8,f5)
Convexity(e1,convex)
Convexity(e2,convex)
Convexity(e5,convex)
Convexity(e6,convex)
Convexity(e7,convex)
Convexity(e8,convex)
Convexity(e3,concave)
Convexity(e4,concave)
EXPORT
e1 as e1, e2 as e2, e3 as e3, e4 as e4, e5 as e5, e6 as e6,
e7 as e7, e8 as e8, f1 as f1, f2 as f2, f3 as f3, f4 as f4, f5 as f5
```

```
END
```

Listing 5.8: Slot definition

```
CREATE TABLE slot AS
 SELECT f1.id as f1, f2.id as f2, f3.id as f3, f4.id as f4, f5.id as f5, e1.id as e1,
        e2.id as e2, e3.id as e3, e4.id as e4, e5.id as e5, e6.id as e6, e7.id as e7,
        e8.id as e8
 FROM   faces AS f1, faces AS f2, faces AS f3, faces AS f4, faces AS f5, edges AS e1,
        edges AS e2, edges AS e3, edges AS e4, edges AS e5, edges AS e6, edges AS e7,
        edges AS e8
 WHERE  e1.edge < e2.edge
        AND e3.edge < e4.edge
        AND f1.face <> f5.face
        AND f1.face <> f4.face
        AND EXISTS (SELECT face FROM face_valency
                    WHERE degree = 4 AND valency.face = f1.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f1.id AND bounds_ef.edge = e1.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f1.id AND bounds_ef.edge = e2.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f1.id AND bounds_ef.edge = e3.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f1.id AND bounds_ef.edge = e4.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f2.id AND bounds_ef.edge = e1.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f2.id AND bounds_ef.edge = e5.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f2.id AND bounds_ef.edge = e8.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f3.id AND bounds_ef.edge = e2.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f3.id AND bounds_ef.edge = e6.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f3.id AND bounds_ef.edge = e7.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f4.id AND bounds_ef.edge = e3.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f4.id AND bounds_ef.edge = e5.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f4.id AND bounds_ef.edge = e6.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f5.id AND bounds_ef.edge = e7.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f5.id AND bounds_ef.edge = e4.id)
        AND EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f5.id AND bounds_ef.edge = e8.id)
        AND EXISTS (SELECT convexity.edge FROM convexity
                    WHERE convexity.type = 2 AND convexity.edge = e1.id)
        AND EXISTS (SELECT convexity.edge FROM convexity
                    WHERE convexity.type = 2 AND convexity.edge = e2.id)
        AND EXISTS (SELECT convexity.edge FROM convexity
                    WHERE convexity.type = 2 AND convexity.edge = e5.id)
        AND EXISTS (SELECT convexity.edge FROM convexity
                    WHERE  convexity.type = 2 AND convexity.edge = e6.id)
        AND EXISTS (SELECT convexity.edge FROM convexity
                    WHERE convexity.type = 2 AND convexity.edge = e7.id)
        AND EXISTS (SELECT convexity.edge FROM convexity
                    WHERE convexity.type = 2 AND convexity.edge = e8.id)
        AND EXISTS (SELECT convexity.edge FROM convexity
                    WHERE convexity.type = 1 AND convexity.edge = e3.id)
        AND EXISTS (SELECT convexity.edge FROM convexity
                    WHERE convexity.type = 1 AND convexity.edge = e4.id)
```

Listing 5.9: Slot definition as SQL

5.5 Theoretical Analysis

This section discusses the imperative algorithm that the query optimizer chooses and discuss its theoretical performance.

5.5.1 Execution Plan

To determine what imperative algorithm the query planner generated, command `ANALYZE` is executed, which gathers statistics about tables and indices, and stores the collected information in internal tables of the database. Then command `EXPLAIN QUERY PLAN` (other DB systems may support different styles of interrogation) is executed. During execution, the query optimizer uses this statistical information for query planning [SQL15a]. The output provides us with information about the execution plan.

It is discovered that for common features (notch, slot, and through-hole) the execution plans chosen by the query optimizer were similar. This enabled us to conclude some execution choices by analyzing typical tasks. Listing 5.10 gives a query fragment for the slot feature; Listing 5.11 shows the corresponding execution plan. The numbers indicated are the table (or its index) lengths estimated by SQLite [Hip13]. These are generated automatically by the SQLite query planner, and change case by case. Valency in the query means the number edges for a face. The testbed was based on SQLite version 3.7.16.2; newer versions of SQLite do not show the estimated table length [Hod14].

```
EXISTS (SELECT valency.face FROM valency
        WHERE valency.degree=4 and valency.face=f1.face) AND
EXISTS (SELECT convexity.edge FROM convexity
        WHERE convexity.type=2 AND convexity.edge=e1.edge) AND
EXISTS (SELECT bounds_ef.edge FROM bounds_ef
        WHERE bounds_ef.face=f1.face AND bounds_ef.edge=e1.edge) AND
```

Listing 5.10: Slot query fragment

```
    0|0|0|SCAN TABLE faces AS f1 (~500000 rows)
    0|0|0|EXECUTE CORRELATED SCALAR SUBQUERY 1
    1|0|0|SEARCH TABLE valency USING AUTOMATIC
            COVERING INDEX (DEGREE=? AND FACE=?) (~7 rows)
    0|1|5|SCAN TABLE edges AS e1 (~250000 rows)
    0|0|0|EXECUTE CORRELATED SCALAR SUBQUERY 2
    2|0|0|SEARCH TABLE convexity USING AUTOMATIC
            COVERING INDEX (TYPE=? AND EDGE=?) (~7 rows)
    0|0|0|EXECUTE CORRELATED SCALAR SUBQUERY 3
    3|0|0|SEARCH TABLE bounds_ef USING AUTOMATIC
            COVERING INDEX (FACE=? AND EDGE=?) (~7 rows)
```

Listing 5.11: Execution plan for slot query

SQLite's three main query optimizations are indexing, reorder joins, subquery flattening. I next discuss what optimizations are adopted by the query planer.

First, consider subquery flattening optimization. Subqueries are multi-block queries (See section 2.4). Sometimes, they can be turned into single block queries by merging any subqueries into the main query body. Such optimization is called subquery flattening in SQLite. It can be determined whether subquery flattening is taking effect from the query plan: if flattening optimization is applied, the execution plan does not show a `SCAN SUBQUERY`: record. Instead, the execution plan shows that the top level query is implemented using a nested loop join of tables. Listing 5.12 shows an example when subquery flattening works [SQL15c].

```
sqlite> EXPLAIN QUERY PLAN SELECT * FROM (SELECT * FROM t2 WHERE c=1), t1;
0|0|0|SEARCH TABLE t2 USING INDEX i4 (c=?)
0|1|1|SCAN TABLE t1
```

Listing 5.12: With subquery flattening

If the flattening optimization is not applied, SQLite executes the subquery and stores the results in a temporary table. It then uses the contents of the temporary table in place of the subquery to execute the parent query. This is also shown in the output of query plan by substituting a `SCAN SUBQUERY` record for the `SCAN TABLE` record. Listing 5.13 gives an example:

```
sqlite> EXPLAIN QUERY PLAN SELECT count(*) FROM (SELECT max(b) AS x FROM t1 GROUP BY a
    ) GROUP BY x;
1|0|0|SCAN TABLE t1 USING COVERING INDEX i2
0|0|0|SCAN SUBQUERY 1
0|0|0|USE TEMP B-TREE FOR GROUP BY
```

Listing 5.13: Without subquery flattening

It is clear that in query plan (Listing 5.11), there is no subquery flattening optimization for `EXISTS` subqueries. Instead, a temporary table is used to cache the data.

Inside each subquery's execution, records show that a covering index is used to access data. This is consistency with the artificial model's scaling performance experiments: the index has some effect on performance while subquery optimization shows little impact.

In the full query plan, the order of range tables is shuffled with respect to the definition, meaning that join ordering optimization is taking effect.

5.5.2 Time Complexity

I now analyze the query plan to estimate its time complexity (Listing 5.11).

The query (5.9) consists of three correlated subqueries: inner queries depend on outer queries, and the inner tables have references to the outer table. Consider the `Valency` query first. The executor executes the outer table scan on faces, taking time $O(f)$ where f is the number of faces, and then executes the inner scan on the valency table using an automatically created covering index. This a temporary index just used in this query to find tuples satisfying subquery predicates. It incurs a cost of $O(f \log(f))$, as the valency table has the same number of entities as the face table, and sorting is needed to make the index.

Then, similarly, the outer query goes through all edge rows, and for each row, searches in an index. This takes time $O(e \log(e) + fe \log(b)))$ where b is the size of the bounds table; the convexity table is the same size as the edge table. However, the bounds table contains $2e$ entries, as each edge has 2 faces, so the overall time is $O(fe \log e)$.

Now, as models get more complex, generally, the individual faces do not get more complex, there are just more of them. Typically, faces have a small fixed maximum number of edges. This observation, used with Euler's formula, means that in complex models, as the number of faces grows, the number of edges approximately grows in proportion, i.e. $O(e) = O(f) = O(n)$ where n is the number of entities in the model.

Overall, then, processing `EXISTS` takes time $O(n^2 \log(n))$: subqueries correspond to outer tables each running an inner scan over a unique index. As $\log(n)$ varies slowly, this explains the quasi-quadratic performance empirically observed.

```
for each face
    index_access(valency), where face=?
    for each edge
        index_access(convexity), where edge=?;
        index_access(bounds_ef), where edge=? and face=?
    end
```

```
end
```

Listing 5.14: Pseudocode for the execution plan

Based on the insights above, the general feature recognition performance can be estimated. The `Bounds_EF` has two inputs (`face` and `edge`) and both have a search domain. In the translation, the nested for-all loop is turned into an algorithm in which the outer loop uses a sequential scan and the inner loop uses index access. The observations suggest that, for most features, `Bounds_EF` and `convexity` are the most important predicates: `Bounds_EF` imposes topological constraints while `convexity` also denotes essential attributes of a feature, for example, the notch feature 3.1 would be a diamond protrusions if `E1` was convex. If considering the most basic situation where the feature is described only using `Bounds_EF` and `convexity`, the algorithm is as given in Listing 5.15:

```
for each face
    for each edge
        index_access(convexity), where edge=?;
        index_access(bounds_ef), where edge=? and face=?
    end
end
```

Listing 5.15: Algorithm for definition using only $Bounds\_EF$ and $convexity$

Following the above reasoning, the complexity is $O(n(n(\log(n)+\log(n)))) = O(n^2\log(n))$. In a more general case, some predicates with input of `face` or `edge` are inserted into the outer loop or inner loop, they will be turned into index based algorithm. Thus, each adding an time complexity of $O(\log(n))$. For example, the case of Listing 5.16 has overall time complexity of $O(n(\log(n)+n(\log(n)+\log(n)+\log(n)))) = O(n^2\log(n))$.

```
for each face
        index_access(face_attribute), where face=?;
    for each edge
        index_access(edge_attribute), where edge=?;
        index_access(convexity), where edge=?;
        index_access(bounds_ef), where edge=? and face=?
    end
```

```
end
```

Listing 5.16: Algorithm example

5.6 Summary and Conclusions

I illustrated how a basic feature recognizer testbed can be devised: a translator that turns feature definitions into SQL queries, using a CAD modeler coupled with a database engine to enable feature recognition in an acceptable time. An advantage of this approach over the similar earlier approach by Gibson is to get “for free” all the insight that has gone into database optimization.

A general and straightforward way is given for declarative level transformations into SQL. The performance are discussed in terms of absolute time and scaling. Experiments show that, as hoped, database optimization provides significant improvements to the time complexity of feature finding, leading to results in an acceptable time. A typical execution plan is given and time complexity of the imperative algorithm is discussed.

However, optimization does not always provide an optimal result: different ways of defining the same feature have different performance even after optimization. It is noted that SQLite is a lightweight database and does not support some advanced features such as recursive SQL and various kinds of advanced indexing. In next chapter, a more powerful PostgreSQL based testbed will be introduced and the benefits of its stronger optimizations will be investigated.

Chapter 6

PostgreSQL Implementation and Linear Performance

6.1 Overview

After the initial experiments with an SQLite testbed, I have developed a fully functional feature recognition testbed according to the conceptual architecture (Fig. 4.1). The database engine is replaced by PostgreSQL with the following reasons:

Firstly, to see whether the optimizations provided by SQLite could be replicated and to determine whether different database engines would arrive at similar query execution plans when used for feature recognition.

Secondly, to further investigate how much performance improvement could be achieved by using a more advanced DB system that has more advanced query optimizations (refer to Tables 2.1 and 2.2).

In this chapter, I will discuss the enhanced feature recognizer (composed of translator, lazy evaluation optimizer, predicate ordering optimizer, DB engine, and executor) based around PostgreSQL and CADfix. This chapter focuses on translation rules, while further details about lazy evaluation and predicate ordering will be discussed in the next chapter.

With reference to the SQLite based testbed, the core question in this chapter is “Can

PostgreSQL DB query optimization help to find features more quickly than SQLite?" with subquestions

1. How should the translator work?
2. What performance is observed in practice?
3. Can the performance be understood by looking at the query plan?

6.1.1 Testbed Implementation Details

The testbed is built around PostgreSQL, which is claimed to be the most advanced open source DB system [od15]. As the architecture (see Fig. 6.1) shows, the recognizer now has a client-server architecture, following how PostgreSQL works. The end-user can send feature manipulation commands, SQL data manipulation commands, or SQL queries from the client terminal to the server. The main part of the code (i.e. the server) turns the SQL query into an efficient algorithm.

Some modules of PostgreSQL are modified and some new modules are added to enable feature recognition. Specifically, they are:

1. modified PostgreSQL's client subsystem to recognize the feature manipulation language,
2. added a translator module, which turns feature definitions into SQL queries,
3. added an LE optimizer module, for lazy evaluation rewriting,
4. added a PO optimizer module, for predicate ordering optimization,
5. added a PO trainer module, for offline training of parameters for lazy evaluation, based on a set of training data,
6. added an importer module, which loads bounding relationships into tables, and

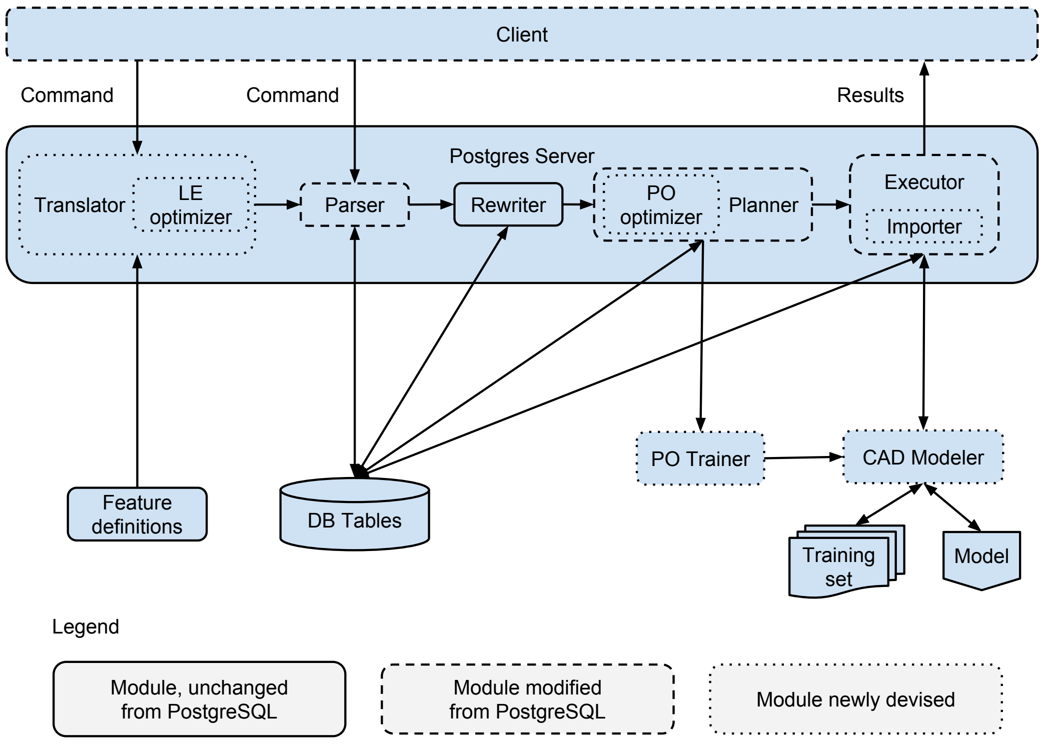


Figure 6.1: PostgreSQL based testbed. LE: lazy evaluation, PO: predicate ordering.

7. added a CAD modeler interface to read models and pass data to the feature recognizer.

The main body of the feature recognizer processes definitions in a similar way to the SQLite testbed. Details are left out here as they are very similar to the methods used in Chapters 4 and 5. Instead, the main differences to the SQLite testbed include:

1. the translator was moved outside the tokenizer and parser (refer to Fig. 5.1), due to the different internal structures of PostgreSQL and SQLite;

2. a new translator with different translation rules to the one in Chapter 5 is implemented, as SQL queries generated by the SQLite testbed have bad performance in PostgreSQL, as explained in Section 6.1.2.

3. lazy evaluation (LE) and predicate ordering (PO) optimization modules (as well as a PO training module) are added into the system. These gives further performance improvements when the CAD modeler has to do extensive numerical computations, as shown in Chapter 7.

I now state the function of the data exchange between different modules:

1. Parser–Feature definitions: look up feature definitions,
2. Parser–DB tables: look up table definitions,
3. Rewriter–DB tables: look up rules/view definitions,
4. Planner–DB tables: look up statistics,
5. Planner (PO optimizer)–PO Trainer: look up predicate ordering parameters,
6. Executor–DB tables: fetch/store user data,
7. Executor–CAD modeler: import CAD data or draw results on views of CAD models.

6.1.2 SQLite Approach Fails with PostgreSQL

Rules are proposed to turn a feature definition into an *existence test subquery* based SQL query in Chapter 5. The SQLite based testbed uses *automatic indexing* (see Section 2.5.4 for how SQLite creates automatic indices) to turn a nested for-all loop into an outer loop using sequential scan and an inner loop using index access 5.5.2. Listing 6.1 gives an example of the translation. Both experiments and theoretical analysis show it gives a quasi-quadratic performance for common features.

```
Definition:         Bounds_EF(e1, f1);
SQL fragment:       EXISTS (SELECT bounds_ef.edge FROM bounds_ef
                    WHERE bounds_ef.face = f1.id AND bounds_ef.edge = e1.id)
Range table:        bounds_ef(edge int, face int);
```

Listing 6.1: Old translation for relation predicates

I reimplemented this translator (to which will be refered as the *old translator*) in the PostgreSQL based feature recognizer to see whether such translation is general enough to work in other DB systems. However, using PostgreSQL, unexpected results were obtained. Even for simple models, PostgreSQL could take days to return results.

The reason is that PostgreSQL has *no automatic indexing* mechanism, and the user has to create any needed indexes manually [IND15]. Thus, PostgreSQL does not process such queries in the same way that SQLite does. PostgreSQL apparently does not have `EXISTS` subquery optimization [ml15] (the reference is rather old, but, no newer document about such optimization was found; I also did not find any related comments in the PostgreSQL source code). Thus, PostgreSQL can only turn the query into nested for-all loops, equivalent to naive translation of the feature definition, with resulting high time complexity.

This led us to rethink the way translation was performed. For flexibility, the translator should work in a way that leads to efficient query processing independent of the choice of the underlying DB engine.

6.1.3 Assumptions

Before giving the new translation approach, I first discuss two assumptions: firstly, assuming that the models input to the feature recognizer are *manifold* solid objects. An n-dimensional manifold requires each point to have a neighborhood to n-dimensional Euclidean space [wik15c]. Solid models are usually restricted to 2-manifold geometry, thus requiring the mathematical neighborhood of each point to be topologically equivalent to a 2-D disk [McM00]. In the B-rep model of a 2-manifold, each edge is shared by exactly two faces, on opposite sides; this is a necessary, but not a sufficient, condition for manifoldness [McM00].

Many classic works on solid modeling were built upon the assumption of manifold models [LG05]. This is because, in a graph based approach, the widely used FAG is suitable for representing a manifold solid model's topology but not those for non-manifold objects [GS98]. To cope with non-manifold objects, Lockett extended the FAG to a mid-surface adjacency graph (MAG) to represent both the faces and edges of the model as nodes on the graph, with the graph arcs representing the connectivity between those faces and edges [LG05]. McMains alternatively showed how to partition a non-manifold into 2-manifold sub-models by splitting non-manifold edges and vertices [MHS01].

The approach depends on the solid model being manifold in the new translation algorithm, as the *full-edge data model* requires an edge to exactly have two adjacent faces. The approach could in principle support non-manifold models by preprocessing them as described above.

Secondly, assuming that the goal is to recognize *connected* features. This means all primitives of the feature are connected locally. For example, in the notch feature(Fig. 3.1), all primitives are connected within the feature. *Disconnected* features have two or more subfeatures that are physically isolated, for example, a pair of parallel faces. When devising this new approach, it was observed that most features of importance are made of sets of connected entities, e.g. machining features such as slots, holes, pockets, etc. These are the most common features in manufacturing or engineering analysis. Unless saying otherwise, in the rest of this Chapter, and Chapter 7, only features of this kind are considered. Chapter 8 will return to how to find more general kinds of features.

6.2 Translation

6.2.1 Data Model

The declarative approach has two modeling steps: firstly, the CAD model features are represented as a text-based feature definition; secondly, the feature definition is then represented textually as an SQL query. Thus, ultimately, the CAD model features are modeled using SQL queries, and it is the translator's job to turn a feature definition into an SQL query.

It is clear that there are various ways to model CAD model features using SQL queries. In Chapter 5, single-column tables are chosen to model the primitive entities in a feature definition. There is a direct mapping from a primitive instance in a feature definition to a single-column table in an SQL query. Although a feature can also be defined using subfeatures, subfeatures are still built from primitives: see for example Listing 3.5. However, queries based on existence test subqueries cannot be directly optimized by PostgreSQL 6.1.2. In fact, SQL queries usually model data using *relations*,: this is the basis of relational database management systems. How to use relations (multiple-column tables) to model feature definitions are explored in this chapter.

I propose to use *full-edge* relations as range tables in SQL queries to model face-edge relationships; a similar approach is used to model edge-vertex relationships. Starting by considering the former, the definition is

$$full\_edge(e, f_a, f_b, convexity) \tag{6.1}$$

In Definition 6.1, e is an edge id of the manifold model, its $convexity$ is denoted in the fourth column (possible values are defined in Table 3.1), and f_a and f_b are the ids of the two faces which are adjacent to e.

The full-edge relation contains rich information, including each edge's local topological information and each edge's convexity. The complex topology of a model can be expressed in terms of connected local topological structures. The $convexity$ information is important for defining a feature: consider for example the notch feature (Listing 3.1). If edge `E1` is convex instead of concave, it becomes a diamond protrusion feature instead. Again, edge convexity information is the only distinction between a square boss and a square pocket. The topology and convexity can be efficiently imported from CAD models without computation by the modeler, as it is stored directly be the B-rep model.

Clearly, a full-edge relation includes more information than a single-column primitive table. Using the latter, subqueries are used to express and test whether a condition is satisfied. The experiments showed that PostgreSQL cannot efficiently process translation based on *existence test subqueries* (see Section 2.4). The advantage of using the full-edge data model is that the constraints a feature must satisfy can be expressed via an *access predicate* or a *filter predicate* (see Section 2.4). Both predicates can be efficiently processed by mainstream DB systems. Section 6.2.2 explains the details.

I next explain the strategy to populate the tables used to find instances of features:

1. The parser sends a command to create the full-edge table as soon as the parser reads any `Bounds_EF` predicates. The importer in the executor then retrieves *all* edge's face adjacency and convexity information from the CAD modeler, creates the tuples $(e, f_a, f_b, convexity)$ and $(e, f_b, f_a, convexity)$, and inserts them into the $full\_edge$ table. Both tuples are stored, the reason is the translator gives an order $(fa\ or\ fb)$ of the two faces when translate the `Bounds_EF` (see Listing 6.5). The order is inherent property inferred from feature definition. Thus, both tuples have to be stored to find the ones satisfy the definition 6.1.

2. All other predicates are translated into arbitrary functions that interact with the CAD modeler and return True or False. When the function is executed the first time, the feature recognizer creates a table. When the function is evaluated for a given id, the function first checks whether the id has a value in the table. If the tuple is empty, the function calls the CAD modeler evaluate the result, and it is stored in the table, otherwise it is returned from the table. This caching is detailed in Chapter 7.

It is noted that the full-edge relation is one possible data model for the translated SQL query, and it is possible to use other relations in CAD models as the range tables. However, the relations must link a *fixed number* of entity instances so that they can be expressed as a table with a fixed number of columns. For example, in a manifold model, an edge has exactly two adjacent faces, so this relation can be expressed using a four-column table $(e, f_a, f_b, convexity)$. In non-manifold models, the number of faces adjacent to an edge is not fixed, so a table with a fixed number of columns can not be used to express such relations.

Another kind of topological relationship is the vertex-edge relation. In the CAD model, each edge has exactly two vertices, so this relation can be modeled using a four-column table $(e, v_a, v_b, convexity)$. If a feature is defined using `Bounds_VE`, edge-vertex relations are loaded in the same way as for face-edge relations. Like the full-edge relations, they can be expressed as access predicates, and filter predicates using similar rules to those in Section 6.2.2. Next subsection will explain how to translate the constraints of `Bounds_EF` using the full-edge range table; feature definitions using `Bounds_VE` can be translated in the same way.

6.2.2 Translation Rules

Again, the goal is to map a feature definition into an SQL query. From the SQLite-based testbed, it is learned that `EXISTS` based subqueries are inefficient as filters. I thus avoid such subqueries by using a filter predicate or an access predicate (see Section 2.4). Some predicates can be more easily mapped to filters than others, and they need to be treated differently according to whether they are relational predicates or attribute predicates.

Relational Predicate Translation Rules

A relational predicate has two (or more) entities, and describes a condition between them. It could be a

1. comparative predicate, for example `Greater_id(id1, id2)`.
2. topological relation predicate, for example `Bounds_EF(e,f)`.

Comparative predicates can be translated in a straightforward way. For example, `Greater_id(id1, id2)` is translated into `id1>id2`. However, topological relation predicates cannot be expressed in such a way as there is no built-in symbol to express `Bounds_EF` or `Bounds_VE`. They are thus turned into access predicates, which can be readily optimised in an SQL query. I next explain how this is done using `Bounds_EF` predicates, and again `Bounds_VE` can be translated using a similar way.

The `Bounds_EF` predicate indicates local neighborhood connectivity of the model. It is one of the most important predicates, and is used in almost every feature definition.

As explained in Section 2.4, the qualification subclause in `WHERE` can be a subquery, an index filter predicate, a table level filter predicate or an access predicate. Using an `EXISTS` subquery results in processing as nested loops in PostgreSQL. As `Bounds_EF` involves *two* entities, it cannot be directly expressed as a filter. Instead, using the full-edge data model 6.2.1, the topological relation predicates are expressed using *access predicates*.

An access predicate specifies two tables are linked together by a key; in feature definitions, for manifold models, it is common that there exist at least two `Bounds_EF` predicates referring to the same `face`. The approach next explained is inspired by the similarity.

In manifold models, each edge is adjacent to two and only two faces. If defining a triplet as *edge-face1-face2*, then the `Bounds_EF` predicate can be replaced as a set of triplets, in which, there always exists a key linking a pair of them. Taking the notch definition for example:

```
        Bounds(E1,F1)
        Bounds(E1,F2)
        Bounds(E2,F2)
        Bounds(E2,F3)
        Bounds(E3,F1)
        Bounds(E3,F4)
        Bounds(E4,F1)
        Bounds(E4,F3)
        Bounds(E5,F2)
        Bounds(E5,F4)
```

Listing 6.2: Bounds constraints in Notch feature

Step1: traverse the `Bounds` constraints and generate triples, in order to discover the link (*key* for access predicate):

```
        T1={E1,F1,F2}
        T2={E2,F2,F3}
        T3={E3,F1,F4}
        T4={E4,F1,F3}
        T5={E5,F2,F4}
```

Listing 6.3: Triplet description

Step2: traverse the triples, generate access predicates.

```
        T1.fa=T3.fa
        T1.fb=T2.fa
        T2.fb=T4.fb
        T3.fb=T5.fb
        T4.fa=T3.fa
        T5.fa=T2.fa
```

Listing 6.4: Access predicates

The transformation above turns relational constraints into access predicates which can be readily optimized in all mainstream DB systems [The15a, Bur10, DuB05]. However, this may be done using an *index access key* or a *hash join key* (see Section 2.4). As shown in experiments, SQLite uses an index access algorithm, giving a quasi-linear performance for common features, while PostgreSQL uses a hash join algorithm, giving linear performance.

The pseudocode is given in Listing 6.5 for the transformation (predicate `Bounds_VE` can be translated in the same way). It has two traversals, the first one to generate triplets and the second one to determine the access predicate keys.

```
//1. Generate Triples from Bounds_EF
for each b[i] in Bounds_EF
    for each b[j] in Bounds_EF
        if b[i].edge=b[j].edge_EF
```

```
            store T(b[i].edge,b[i].face,b[j].face)
            store T(b[i].edge,b[j].face,b[i].face)
        end
    end
end
//2. Generate relation list
for each T[i] in triples
    for each T[j] in triples
        if T[i].fa=T[j].fa
            output access predicate T[i].fa=T[j].fa
        else if t[i].fa=t[j].fb
            output access predicate T[i].fa=Tt[j].fb
        else if T[i].fb=T[j].fa
            output access predicate T[i].fb=T[j].fa
         else if T[i].fb=T[j].fb
            output access predicate T[i].fb=T[j].fb
         end
    end
end
```

Listing 6.5: Pseudocode for relational predicates transformation

In practice, it is possible a feature is defined with an edge with only one face, users can make the following processing: add an extra `Bounds_EF(e4,f_generated_xxxx,e4)` to the definition before translating it, where `f_generated_xxxx` is some arbitrary unique name. When finding features, and report them, do not report (or draw) and faces with names `f_generated_xxxx`.

Attribute Predicate Translation Rules

Attribute predicates have only a *single entity* and some *attribute constraints*. Listing 6.6 gives two example:

```
Face_has_number_of_edges(face:f, int:imin, int:imax)
Face_contained_in_box(face:f, box:b)
```

Listing 6.6: Example attribute predicates

In practice, some attribute predicates are commonly used in almost all feature definitions; an example is edge convexity: see for example, Listings 3.4 and 3.5. Others are rare and used only in special tasks. They may take much longer time to compute, for instance, finding a feature for which the area of a particular face lies in a certain range. They are treated separately. Edge convexity information are stored in the full-edge tables, and turn these predicates into filters. For example:

```
Definition:       Convexity_is(edge, convex)
SQL fragment:     AND full_edge_e.convexity=convex
```

Such a *filter* might be treated as an *index filter predicate* or a *table level filter predicate* (see Section 2.4), based on statistical analysis performed by the query planner.

Other unusual or slow attribute predicates are turned into Boolean built-in functions. These functions (or remote predicates) call the CAD modeler to do the calculation and return True or False. For example:

```
Definition:       Face_area_in_range(f, vmin, vmax)
SQL fragment:     AND  face_area_in_range(f, vmin, vmax)
```

Such function translation makes us ask the following questions:

1. If functions are computed for all possible inputs, and they are slow, the performance is obviously decreased. Can the work load be reduced to improve the performance?

2. Can the idea of caching be used to avoid recreated function calls?

3. if there are multiple filters, the order in which they are applied may affect performance a lot. The DB engine does not have statistics needed to optimize this. Can they be obtained and provided to the query planner to generate a better execution plan?

The answers to the above questions, involve use of lazy evaluation and predicate ordering optimization, as will explained in the next chapter. In this chapter, I mainly focus on *basic local* features which can be described by relational predicates and edge convexity attributes, such as notches, slots, etc.

Full Definition Translation

Having explained how each kind of predicate is translated, I now show to generate the full query in Listing 6.7. I assume that only `Bounds_EF` bounding is used; features defined using `Bounds_VE` can be translated similarly. This approach only works for local features composed of connected faces and edges, without isolated entities. Most manufacturing features required in engineering are this type [SW95, HPR00]. I will give an architecture for a stand-alone feature recognizer in Chapter 8, which is general enough to recognize non-local features.

The pseudocode in Listing 6.7 also does not cover features that are defined by subfeature. Because usually their translation is rather straightforward: firstly, the subfeatures defined by `Bounds` are translated using algorithms described in Listing 6.7; secondly, translate the feature using subfeatures. In the translation, the target list subclause is translated from the `EXPORT` statement of the definition, the relation list subclause is generated using the subfeature tables and the qualification subclause is generated by straightforwardly translating the attribute constraints of the these subfeatures into filter predicates.

```
set query="CREATE TABLE <feature> AS";
set target_list_subclause="SELECT";
set relation_list_subclause="FROM";
set qualifications_subclause="WHERE";
//transform bounds predicates
[APS,FS,ES]=transform_bounds_ef(definition);
    APS: access predicate set;
    FS: set of {f_i, full_edge_e_m.f_a/b} where f_i is the face id in
            definition, its internal name is full_edge_em.f_a/b;
    ES: set of e_m, full_edge_e_m_edge where e_m is the edge id in
            definition, its internal name is full_edge_e_m.edge;
//transform edge convexity predicates
[ECPS]=transform_edge_convexity(definition, ES);
    ECPS: edge convexity predicate set;
//transform other attribute predicates
[OAPS]=transform_attributes(definition, FS, ES);
    OAPS: other attribute predicates set;
//generate target list, relation, qualification subclause
qualifications_subclause=append(qualifications, APS, ECPS, OAPS);
relation_list_subclause=append(relation_list_subclause, ES);
target_list_subclause=append(target_list_subclause, full_edge_e_m.f_a/b  AS f_i,
    full_edge_e_m.edge  AS  e_m;
//complete the query
query=append(query, target_list_subclause,relation_list_subclause,
    qualifications_subclause);
```

Listing 6.7: Full query translation

Firstly, traversing relational predicates, producing three result sets: an access predicate set, a face-name set describing the face ida in the definition, and a similar edge-name set. As explained in Section 6.2.1, each face and edge has an internal name that is used to express the SQL query. The face-name set and edge-name set give the internal names of entities in the feature definition. Listing 6.5 generates the APS in the function $transform\_bounds\_ef$.

Secondly, producing convexity predicates using the edge-name set and the definition.

Thirdly, traversing other attribute predicates, producing *functions* using the face-name set and edge-name set.

Fourthly, producing the relation list, target list and qualification clause separately (see Section 2.4 for definitions).

Finally, concatenating the three clauses to generate the final query.

An example is shown for the notch feature in Fig. 3.1. Using the new translation rules, the query generated is:

```
CREATE TABLE notch AS
SELECT full_edge_e1.edge AS e1, full_edge_e2.edge AS e2, full_edge_e3.edge AS e3,
       full_edge_e4.edge AS e4, full_edge_e5.edge AS e5, full_edge_e1.face1 AS f1,
       full_edge_e1.face2 AS f2, full_edge_e2.face2 AS f3, full_edge_e3.face2 AS f4
FROM   full_edge full_edge_e5, full_edge full_edge_e4, full_edge full_edge_e3,
       full_edge full_edge_e2, full_edge full_edge_e1
WHERE full_edge_e1.face2=full_edge_e2.face1
  AND full_edge_e1.face1=full_edge_e3.face1
  AND full_edge_e1.face1=full_edge_e4.face1
  AND full_edge_e1.face2=full_edge_e5.face1
  AND full_edge_e2.face2=full_edge_e4.face2
  AND full_edge_e2.face1=full_edge_e5.face1
  AND full_edge_e3.face1=full_edge_e4.face1
  AND full_edge_e3.face2=full_edge_e5.face2
  AND full_edge_e1.convexity=1 AND full_edge_e2.convexity=2
  AND full_edge_e3.convexity=2 AND full_edge_e4.convexity=2
  AND full_edge_e1.face1 < full_edge_e1.face2
  AND full_edge_e2.face2 < full_edge_e3.face2
  AND full_edge_e3.face2<>full_edge_e2.face2
  AND full_edge_e3.face2<>full_edge_e1.face2
  AND full_edge_e3.face2<>full_edge_e1.face1
  AND full_edge_e2.face2<>full_edge_e1.face2
  AND full_edge_e2.face2<>full_edge_e1.face1
  AND full_edge_e1.face2<>full_edge_e1.face1
  AND full_edge_e5.edge<>full_edge_e4.edge
  AND full_edge_e5.edge<>full_edge_e3.edge
  AND full_edge_e5.edge<>full_edge_e2.edge
  AND full_edge_e5.edge<>full_edge_e1.edge
  AND full_edge_e4.edge<>full_edge_e3.edge
  AND full_edge_e4.edge<>full_edge_e2.edge
  AND full_edge_e4.edge<>full_edge_e1.edge
  AND full_edge_e3.edge<>full_edge_e2.edge
  AND full_edge_e3.edge<>full_edge_e1.edge
  AND full_edge_e2.edge<>full_edge_e1.edge
```

Listing 6.8: Notch query: new translation

6.3 Discussion

With the declarative approach, a feature is described using a set of constraints and finding the feature requires testing all entity combinations to generate some sets of entities that are candidate target features. However, the constraints have different roles in the feature definition. The constraints in Listings 3.2–3.3 can be classified into the following six categories (again, I also only consider locally connected features):

1. Bounds constraints
2. Edge convexity constraints
3. Entity uniqueness constraints
4. Entity id rank constraints
5. Geometric constraints such as face shape
6. Other numerical constraints such as area, length, angle, etc.

Bounds constraints are local neighborhood relationship conditions. Local neighborhood adjacency relations are the most common constraints in feature definitions. Many features can be described using a set of `Bounds_EF` predicates (and possibly other predicates). In this Chapter, bounds constraints are turned into access predicates which can readily be optimized in mainstream DB systems.

Edge convexity is used to describe many manufacturing features [HPR00]. For example, convexity is a necessary constraint to tell a square pocket from a square boss; or it could be used to distinguish whether a feature is open or blind.

Entity uniqueness constraints are generated automatically by the translator, in order to reject result sets that may satisfy other constraints but are not able to construct a meaningful solid. For example, in Fig. 6.2, without entity uniqueness constraints, the circle loop C1 on the side face of the hole also obeys the hole definition that is defined

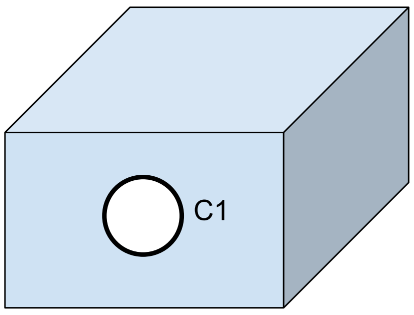


Figure 6.2: Circle recognized as a hole feature

just using bounds, edge convexity and face geometry constraints. In fact, this issue arises in almost all feature definitions. The solution is to assume that, in most cases, if a feature definition mentions e.g. two faces `f1` and `f2`, it is the engineer's intent that they should be distinct. Thus, the translator automatically inserts SQL fragments like `f1<>f2` into the final query for each *pair* of entities of the same kind. The user no longer needs to write `Different_id` clauses as in the SQLite testbed, as they are now automatically generated.

In practice, users are allowed override this assumption by adding clauses of the form `ALLOWING f1=f2` to state that some particular entities may be the same. For example, if the task is to find a through hole in a sphere, the uses has to add this statement into Definition 5.6, otherwise, no results will be returned.

Entity rank constraints refers to entity ids, for example, `Greater_id`. They can be used to remove topologically symmetric results. As source data (the full-edge range tables in Section 6.2.1) has two tuples $(e, f_a, f_b, convexity)$ and $(e, f_b, f_a, convexity)$ for each edge are imported. In the result, both tuples (e, f_a, f_b) and (e, f_b, f_a) are kept—this is a topological symmetry. The logic leads to repeatedly finding the same solution in which the names of the entities are permuted. For example, see the notch in Fig. 3.1: interchanging the roles of faces `f1` and `f2`, and `f3` and `f4` (as well as

Table 6.1: Topological symmetry result for notch feature

Feature primitives	e1	e2	e3	e4	e5	e6
Feature 1	14	13	17	18	16	15
Feature 2	13	14	17	18	16	15
Feature 3	13	14	15	16	17	18

various edges) gives another interpretation of the same notch. Table 6.1 gives actual output for finding a single notch feature if the feature is defined without entity rank constraints. The first row in the table labels the primitives of the feature; rows 2–4 show three features have been found, where each column gives the id of some edge primitive. Clearly, features 1–3 are the same feature, except with different assignments of actual faces to those in the definition.

Typically, it is more efficient not to generate such toplogically symmetric results than to remove them later. One way to do this is to give a direction to the graph representing the feature. For example, if the user adds `Lower_id(f1, f2)`, it will prevent notch features from being reported twice. However, it is still not clear how to automatically generate these constraints. I give the control to end-users, and it remains as future work to explore an automatic approach.

Other numerical constraints such as area, length, angle, etc., are specially useful for engineering analysis features. They are translated into arbitrary functions, whose numerical output is cached in order to save recomputing it if it is needed again later. Relevant details will be explained in Chapter 7.

6.4 Experiments

The SQLite based testbed achieved approximately $O(n^2)$ performance for basic features (notch, slot, through-hole) [NMS+15]. However, when the database engine is replaced by PostgreSQL, still using the original translation strategy, the performance

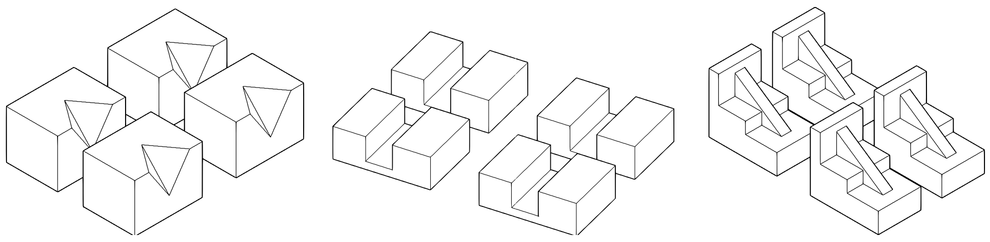

(a) Notch (b) Slot (c) Step-rib

Figure 6.3: Artificial models for performance testing

was much worse; indeed no feature finding results were returned in any reasonable time. Analysis of the cause led to the new translation approach given here. The questions below are considered:

1. Does the new approach perform better than the previous approach? If so, why?
2. Can the new approach be optimized by both DB optimizers? If they do not give the same improvement, why?

In order to determine scaling performance, The same test family models as used for the SQLite testbed are used: the models comprise an increasing number of blocks (2^n where $n = 0, \dots, 11$). Notch, slot and step-rib feature are used here. Such models can give how the performance scales when models increase in complexity in a regular way. Fig. 6.3 shows the models for $n = 2$.

The old and new translation approaches using the same database engine (SQLite) are compared first; the experiments show that the expected improved computational complexity is observed. Secondly, the relative performance of two different database engines (SQLite and PostgreSQL) are compared.

The test platform used in Chapters 6 and 7 is described in Table 6.2.

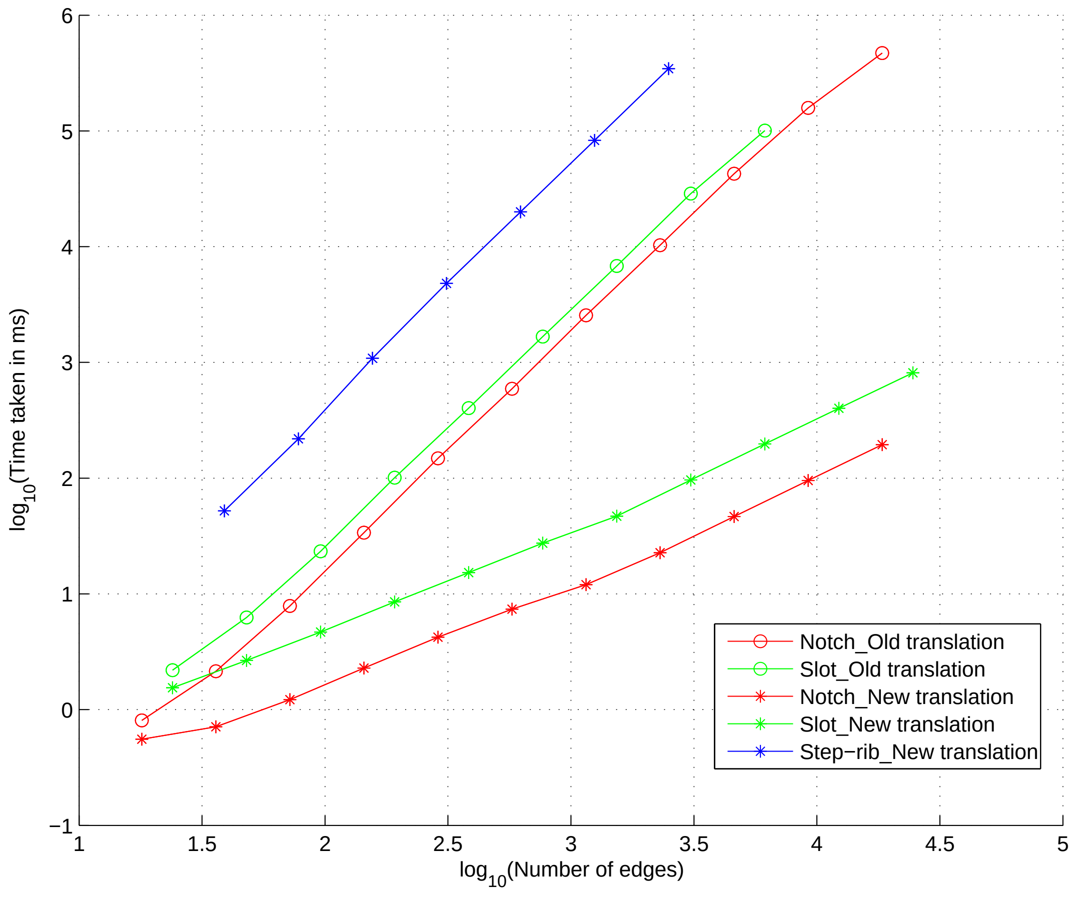


Figure 6.4: Performance comparison between new and old translation using SQLite.

Table 6.2: Test platform

CPU	Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz
Memory	32GB
OS	Debian GNU/Linux 8.0 (jessie)
Compiler	GCC 4.9.2
PostgreSQL version	3.7.16.2

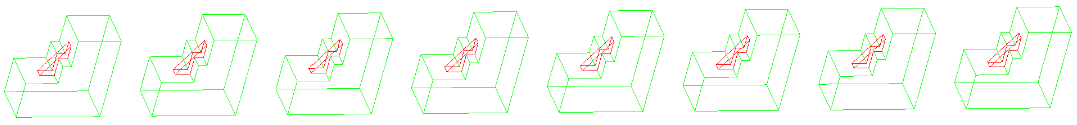


Figure 6.5: Features are drawn on the CAD model to valid the result

Translation Approach	Notch	Slot	Step-rib
Old	1.98	1.98	—
New	0.89	0.90	2.12

Table 6.3: Exponent of performance of old and new translations using SQLite

6.4.1 Old and New Translation Using SQLite

The old and new translators with SQLite are compared; the former beings the older *existence test* subquery translation based approach, and the latter being the newer *access predicate* translation described in this chapter. The translators turn the same definition into different queries, and the experiments here aim to give which one achieves better performance.

Three kinds of models –notch, slot and step-rib feature recognition are performed on the artificial family model set. Features are drawn on the CAD model to valid the result as Fig. 6.5 shows. The log-log plot is given in Fig. 6.4 which gives the time taken in milliseconds to find all features of the given type in each model, versus the total number of edges in that model (step-ribs took too long to find using the old approach, so no results are presented in that case). Performance in the log-log plot approximately follows a straight line relationship in each case, indicating that time taken to find features is reasonably modeled as $t = \alpha n^p$ where p is the slope of the line, and n is the number of entities. (As noted earlier, the number of edges is roughly proportional to the number of entities). In practice, in order to obtain the asymptotic behavior of the algorithms (for larger models), the slope past the point at which the slope seems to stabilize are measured. The slopes are given in Table 6.3.

Database Engine	Notch	Slot	Step-rib
SQLite	0.89	0.90	2.12
PostgreSQL	0.91	0.94	0.95

Table 6.4: Performance of new translation using SQLite and PostgreSQL

From the time complexity table, it is clear that, although both translations are effectively optimized by the database engine, the computational complexity is quite different. The old approach results in approximately $O(n^2)$ performance for notch and slot features. For step-rib features the system failed to return results in an acceptable time—step-ribs contain many more entities (9 faces and 12 edges) than notches (4 faces and 5 edges) or slots (5 faces and 8 edges). In contrast, the new translation approach results in roughly linear performance for notch and slot features, and approximately quadratic performance for step-rib features.

6.4.2 New translation using SQLite and PostgreSQL

Next, the new translator with both SQLite and PostgreSQL are used to answer the question of for the same SQL query, which engine performs better and how the query is executed. The same families of models are used and the corresponding slopes are given in Table 6.4. Approximately linear complexity is achieved using PostgreSQL, for all cases.

This result is significant, as it implies that a system based on these ideas should scale to very large industrial models. As far as I know, no other published feature finder displays linear performance; indeed many papers note the exponential complexity of graph based feature finders [SAKJ01].

Both query optimizations in SQLite and PostgreSQL give a nearly linear performance for the simplest features, but SQLite can exhibit worse performance for more complex features. To further understand why PostgreSQL achieves linear performance for

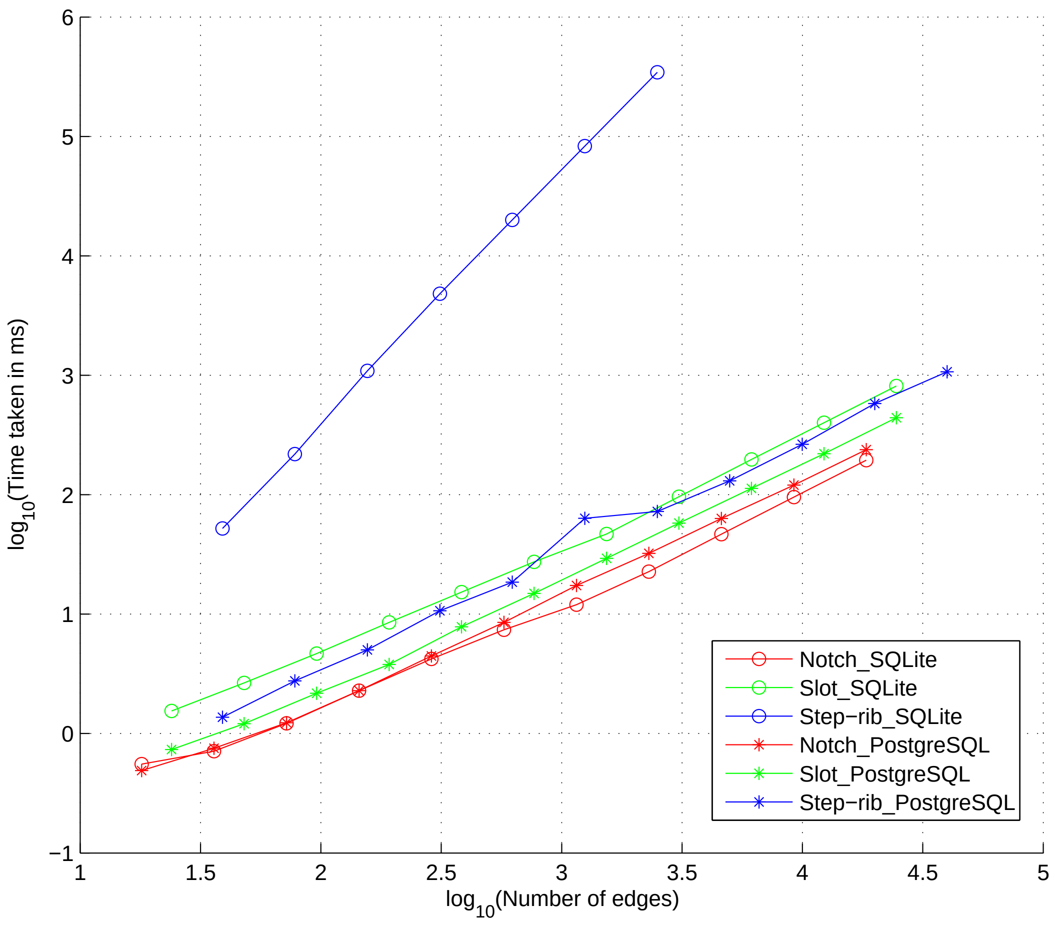


Figure 6.6: Performance of new translation using SQLite and PostgreSQL

step-ribs while SQLite does not, the optimizations used by each database engine are analyzed. They are quite different.

Firstly, considering how SQLite processes the query. Fig. 6.7 shows part of a typical SQLite query plan for slot feature recognition. SQLite optimizes the query mainly by the use of automatic covering indexes, and no changes are made to the order of joins. As temporary index creation requires sorting, the time taken must be at least $O(n \log n)$. Detailed considerations of the query plans reveal that although notch, slot, and step-rib features all use a covering index, they are used quite differently. For step-ribs, execution steps like the below are used:

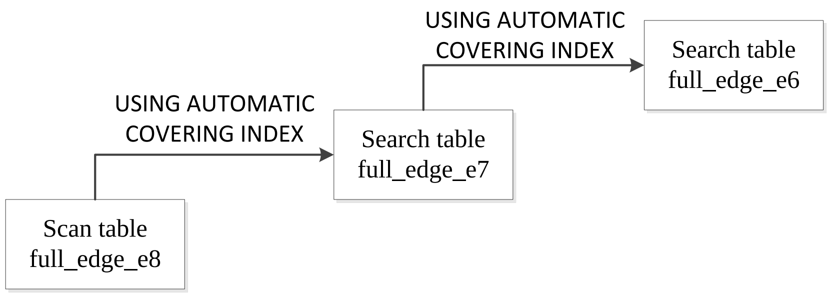

USING AUTOMATIC COVERING INDEX
Search table full_edge_e6
USING AUTOMATIC COVERING INDEX
Search table full_edge_e7
Scan table full_edge_e8

Figure 6.7: New translation query plan in SQLite

```
0|0|11|SCAN TABLE full_edge AS full_edge_e1 (~50000 rows)
0|1|0|SEARCH TABLE full_edge AS full_edge_e12 USING AUTOMATIC COVERING INDEX (
    convexity=?) (~7 rows)
```

where table full_edge is defined as

```
full_edge(edge INTEGER, face1 INTEGER, face2 INTEGER, convexity INTEGER)
```

While SQLite processes convexity using a covering index, almost all tuples satisfy the convexity constraint, so the result is almost a sequential scan of all tuples, leading to $O(n^2)$ overall performance. This is not the case for notch and slot features. Experiments show that if creating face and edge indexes explicitly, SQLite can also achieve quasi-linear performance for step-ribs.

Let us now consider the execution plan used by PostgreSQL, as illustrated in Fig. 6.8. Here, first the order of range tables is shuffled allowing join re-ordering optimization. Tables are accessed sequentially before pairs are jointly processed by hash joins. The simplest kind of hash join includes two steps: first the smaller relation is used to construct a hash table, and then the larger relation's tuples are used to probe the hash table to find matches. To understand the performance, consider the simplest situation: two (unindexed) relational tables, both with $O(n)$ tuples. The cost is composed of four linear components: reading the inner table, hashing the inner table, reading the outer table, and probing the hash table, giving a total cost of $O(n)$. This expectation is verified in the experiments. The PostgreSQL query optimization is more powerful and the

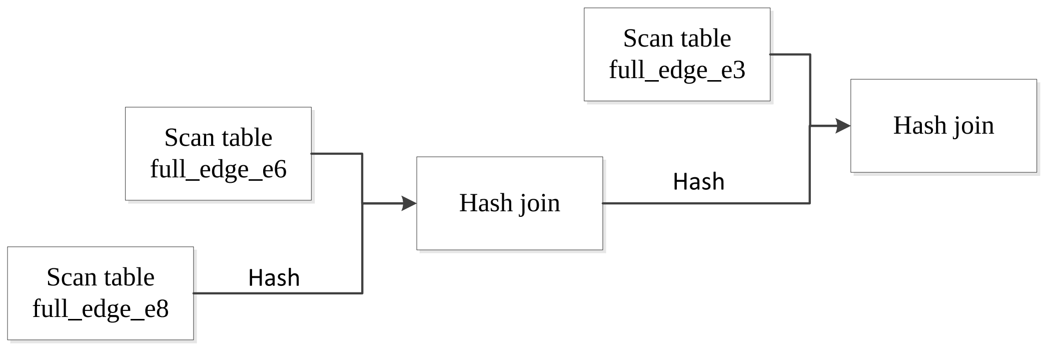


Figure 6.8: New translation query plan using PostgreSQL

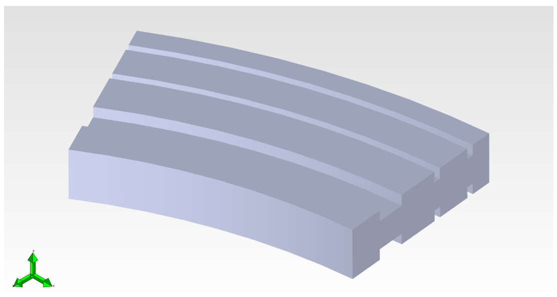


Figure 6.9: Carbine

significant result is that simple features can now be found in *linear* time.

6.4.3 Real World Performance

Real industrial CAD models are more challenging: models may include hundreds of thousands or even millions of entities. In this case, performance is potentially a serious problem. In real models, there are more types of entities, including subfeatures, and features are more complex than the simple ones used in earlier tests. All of these are big challenges for traditional algorithms. In this section, I show tests on several larger models to help assess the potential of the approach for industrial use.

Firstly, the performance of the new approach with the previous work are compared, using increasingly complex models of a carbine, switch and CPU heat sink (see Figs. 6.9–

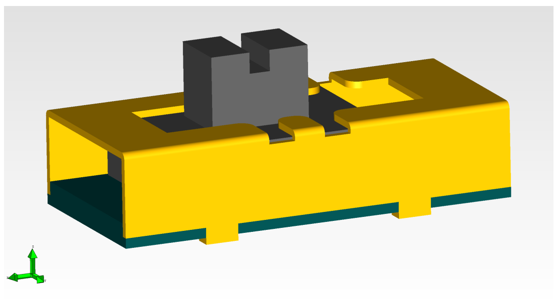


Figure 6.10: Switch

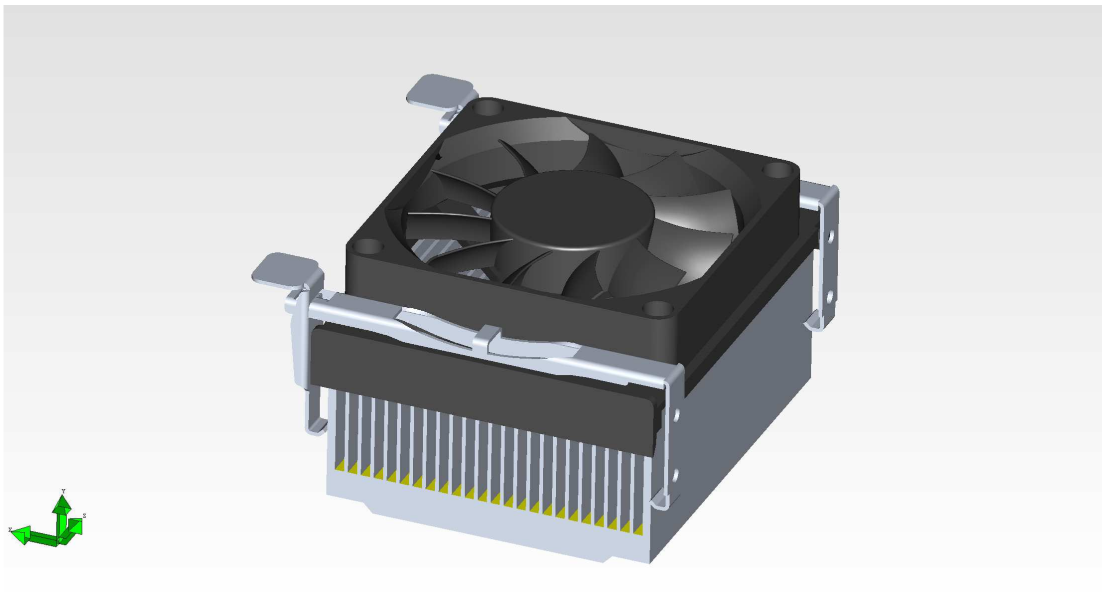


Figure 6.11: CPU heatsink

6.11); the features to be found were again open slots, blind slots, and through holes.

Feature finding (see Table 6.5) took much less time than when using the previous approach for the CPU heat sink and switch. Similar times were achieved for the carbine, probably due to its simplicity. This is in agreement with the earlier experimental finding that the new approach has lower time complexity—it scales up better to larger models. These results are very encouraging, and show that the current approach can rapidly find features in models of realistic complexity. Feature finding took just 0.1 s even for the heat sink, which has over 2000 edges.

Model	Carbine	Switch	CPU heatsink
Number of edges	84	330	2388
Number of slots	6	9	24
Unoptimized query	15 hours	-	-
Old translation (SQLite)	50 ms	220 ms	6940 ms
New translation (PostgreSQL)	47 ms	69 ms	107 ms

Table 6.5: Time taken to find slots in real models

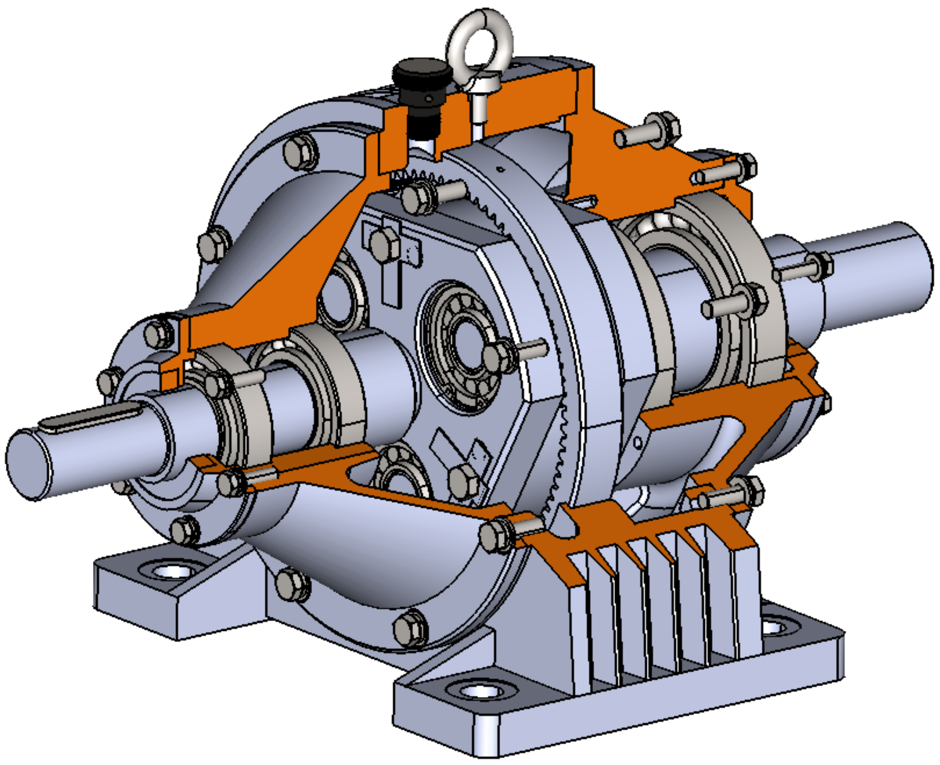


Figure 6.12: Reducer

It is also performed that further experiments on real industrial models to assess performance. Fig. 6.12 shows a moderately complex reducer model obtained from [Gra15], with 17774 edges. It includes hundreds of open slots, blind slots, through-holes, and other features. The feature recognizer can find such features in this model in a fraction of a second: see Table 6.6.

Feature	Open slot	Blind slot	Throughhole
Number of features	140	146	164
Time taken	168 ms	176 ms	87 ms

Table 6.6: Time taken to find various features on a reducer model with 17774 edges.

More complex features can be defined using subfeatures—features can often be decomposed into several similar sub-structures. Finding such substructures first and then combining them into a complete feature simplifies the writing of feature definitions. For example, if defining an adjacent-pair-of-blind-slots feature, and seeking it in the reducer model. This new feature comprises two round corner blind slots that are connected by short edges. It can be first to find the slot features (with 17 edges and 10 faces), and then to determine which of those are adjacent and connected by short edges. It takes 168ms to find all 5 round corner blind slots and another 56ms to find pair-slot features. More generally, however, regular features with *arbitrary* numbers of elements, such as a ring of holes, a gear, or a row of slots, are most easily defined recursively. This in turns needs a database that can handle recursive SQL queries; I intend to investigate such an approach in the future work.

6.5 Summary

It is discovered that SQLite and PostgreSQL take very different approaches to query optimization. The old translator turns constraints into existence test subqueries that might not be very efficiently evaluated, while the new translator processes predicates differently: relational predicates are turned into access predicates while others are turned into filter predicates or Boolean functions.

It is discussed that how to write an effective feature definition and how the constraints are transformed equivalently.

It is shown that the significant result using the new translator can find simple features in *linear* time on PostgreSQL or approximately linear time on SQLite. It is also explored that the execution plans and learned that for the same access predicates, SQLite uses an index to access data of interest while PostgreSQL uses a hash join algorithm to reduce the search space for optimization.

Chapter 7

Further Improvements Using Lazy Evaluation and Predicate Ordering

7.1 Overview

The new translator turns relational predicates (in feature definitions) into access predicates (in SQL queries) and expensive attribute predicates into Boolean function calls to the CAD modeler to do the calculation. There are well-developed DB algorithms to handle access predicates or filters (see Section 2.4). However, the attribute predicates are not transparent to the query optimizer, and thus cannot be optimized directly. Such functions make us consider the following questions:

1. If the attribute predicates are evaluated for all tuples in the range table, the performance will be obviously slow if the predicates are expensive to compute. Can the number of function evaluations be reduced to improve the performance?

2. Can the idea of caching be applied to avoid repeated evaluation of the same function results?

3. If there are multiple attribute predicates acting as filters, the execution order may (or may not) have a strong effect on performance. The DB engine does not have statistics allowing it to optimize this order. Can they be obtained and provided to the query planner to generate a better execution plan?

The work is extended using lazy evaluation and predicate ordering optimizations to answer these questions. By introducing lazy evaluation, expensive functions are evaluated only when they are definitely needed, avoiding unnecessary computing. As well as lazy evaluation together with the idea of caching are adopted. Specifically, the computed numerical results calculated by the attribute functions are cached, so that during each call to find features, the expensive functions are executed only when a local table does not already have the corresponding results, avoiding repeated function calls. By automatic predicate reordering, the function calls are executed in order with minimal cost, achieving a further performance improvement.

As the PostgreSQL based feature recognizer testbed architecture (Fig. 6.1) shows, the lazy evaluation optimizer is part of the translator. When it reads in a feature definition, it firstly rewrites the query to permit use of lazy evaluation (still as an SQL query). Specifically, it moves the attribute functions into the `HAVING` clause of the query, as shown in Listing 7.8. Next, the SQL query is transformed into an algorithm by the query planner. When the algorithm is executed, the functions in the `HAVING` clause are evaluated on the target list (see Section 2.4) which only contains those results satisfying all constraints in the `WHERE` clause. Lazy evaluation is achieved by rewriting the query based on the approach above. Thus, it is an optimization at the declarative level.

On the other hand, the predicate ordering optimizer uses statistics concerning the attribute predicates and selects the optimal order in which to apply these as filters. such predicate ordering is part of the query planner and is a procedural level optimization.

In this chapter, I first explain the theory and then give experiments to illustrate the performance improvements achieved. It is discovered that both are of greatest benefit when finding complex features that involve more than `Bounds` and edge convexity predicates.

7.1.1 Lazy Evaluation

The idea of lazy evaluation is to avoid computing things until the very moment that they are definitely needed—there is no point in computing things that may later turn out to be unnecessary. For example, suppose `p(x)` and `q(y)` are predicates, which may be expensive to evaluate. Consider the expression `p(x) AND q(y)`. Both will be evaluated and the final result is computed using logical `AND`. However, if `p(x)` is False, the overall expression must be False, and `q(y)` do not to be computed at all, saving unnecessary work. (This assumes that the predicates have no side-effects).

Lazy evaluation can help to ensure that the predicates are only evaluated on a small candidate set. For example, when finding features such as *small* pockets, only the areas of faces definitely belonging to pockets are needed to be computed. In the feature finder, predicates are evaluated at runtime either by local lookup in cache tables, or remotely by the CAD modeler; sometimes the latter may take a long time.

Lazy evaluation is realized in the system by steps in the translation stage: expensive predicates are expressed as foreign SQL functions and evaluated during execution, by calling the CAD modeler directly.

The following property of SQL queries (also explained in Chapter 2) are used: predicates placed in `WHERE` clauses are evaluated on *all* tuples of the range tables, while predicates placed in `HAVING` clauses are only evaluated on temporary results that fulfil the conditions in the `WHERE` clauses. Thus, the translator puts potentially expensive predicates into `HAVING` clauses for efficiency. The only predicates placed into `WHERE` clauses are relational predicates (which can be optimized as access predicates) and edge convexity attribute predicates (which work as filters).

Furthermore, memoization (caching) is used: each time, when evaluating the function that is translated from the predicate, first to see if it is already available in a local table. If not, a remote call is made to CADfix to calculate the result, which is then also placed in the local table. Caching it can save recomputing the result if it is needed again later.

Table 7.1: Caching tables for constant attribute predicates

Predicate	Table required	* meaning
*_valency	*_valency(* int, degree int)	face, vertex
Face_has_number_of_*	face_has_number_of_* (* int, number int)	vertices, edges
Body_has_number_of_ *	body_number_of_*(body int, number int)	vertices, edges, faces
*_has_geometry	*_has_geometry(* int,*type int)	Edge, Face

Similar gains are provided for other feature definitions involving expensive predicates.

In practice, all attribute predicates except `convexity` are translated into functions that are lazy evaluated. The attribute predicates in Listing 3.2 and Listing 3.3 are classified into three categories based on how they are translated and how the data are cached.

- constant attribute predicates, which are about constant topological or geometric attributes. They are translated into the same name functions, which use the same name two-column tables $(id, attribute)$ to memorize the computed values. For example, the predicate `Face_has_number_of_edge` requires us to cache the boundary edge number of faces into the table $face\_has\_number\_of\_edges$ $(face\ int, number\ int)$. Once the predicate is evaluated, it first checks whether the edge numbers are cached in the $face\_has\_number\_of\_edges$ table, if existing, it tests whether it is equal to the value that the user defined and returns True or False correspondingly; else it computes the edge numbers first, caches it into the table $face\_has\_number\_of\_edges$ and then compute the True or False; Table 7.1 gives the caching tables for constant attribute predicates;
- range attribute predicates, which test whether the attribute of an entity is in a certain range. These predicates are translated into the same name functions; they also use the same name two-column tables $(id, attribute)$ to memorize the

computed values. For example, the predicate `Face_contained_in_box` requires us to cache the bounding box of faces into the table $face\_box(face\ int, box\ box)$. Once the predicate is evaluated, it first check whether the bounding box is cached in the $face\_box$ table, and if existing, it checks whether the bounding box of the face is inside the user defined box and returns True or False correspondingly, else it returns the bounding box first and caches it into the table $face\_box$; and then compute the True or False. The range attribute predicates are usually more computation intensive than constant attribute predicates. Thus, they have better performance improvement using lazy evaluation and caching. Table 7.2 gives the caching tables for range attribute predicates;

- miscellaneous predicates, are mainly about distance and angle related constraints. These predicates are translated into the same name functions. However, they may use multi-column tables to memorize the computed values. For example, the predicate `Cylinder_axis_aligned_within` requires us to cache the axis of the cylinder and the angle with a vector into the table $Cylinder\_axis\_angle$ $(face\ int, axis\ vector, v\ vector, angle\ real)$. Once the predicate is evaluated, it first checks whether the axis is cached in the $Cylinder\_axis\_angle$ table, if existing, it checks whether the axis of the cylinder face aligned the user defined vector within a certain angle and returns True or False correspondingly, else it computes the axis and the angle with the vector and caches them into the table $Cylinder\_axis\_angle$; and then checks the True or False. The system builders may need to tailor a table used for caching for more sophisticated predicates. Table 7.3 gives the caching tables for miscellaneous predicates;

7.1.2 Predicate Ordering

The other (optional) optimization is predicate ordering. This is also used to reduce the workload performed by CAD modeler. In lazy evaluation, expensive calculations

Table 7.2: Caching tables for range attribute predicates

Predicate	Table required	* meaning
*_contained_in_box	*_box(* int, box box)	Vertex, Edge, Face, Body
*_contained_in_uvbox	*_uvbox(* int, uvbox uvbox)	face
*_in_range	*(edge int, length real)	Edge_length, Face_area, Body_volume
*_radius_in_range	*_radius(* int, radius real)	Sphere, Cylinder, Cone_min/max, Torus, Ellipsoid
*_than	*(edge int, length real)	Edge_longer, Face_larger

Table 7.3: Caching tables for miscellaneous predicate

Predicate	Table required	* meaning
Vertex_near_*	vertex_*_distance (vertex int, * int, distance real)	vertex, edge, face
*_centre_near	*_point_distance(face int, p point, distance real)	Sphere, Torus
*_aligned_within	*_angle(face int, * vector, v vector, angle real)	Plane_normal, Cylinder_axis, Cone_axis, Ellipsoid_axis, Torus_axis

are performed on the result set, but the order of the calculations affects the overall time taken. For example, suppose the feature is defined as “all faces satisfying p_1 and p_2”, where p_1 and p_2 are two predicates. Let us assume for example that it costs 3 time units ($3U$) to evaluate p_1, and it costs 10 time units ($10U$) to evaluate p_2 (for simplicity, assuming this is constant for each entity—in relaity it will depend on the data). Secondly, suppose that typically, p_1 is true 10% of the time and p_2 is true 50% of the time. Both these relative costs, and the probability of rejection by the predicate when used as a filter, may be estimated by evaluating the predicate for a training set of models, as will be explained later.

Here are various query plans, for this example, together with the estimations of how long each takes.

```
s1 = Select from all faces those satisfying p1.
s2 = Select from all faces those satisfying p2.
result = s1 intersect s2.
```

Listing 7.1: Evaluation 1

```
s1 = Select from all faces those satisfying p1.
result = Select from s1 those satisfying p2.
```

Listing 7.2: Evaluation 2

```
s2 = Select from all faces those satisfying p2.
result = Select from s2 those satisfying p1.
```

Listing 7.3: Evaluation 3

The approach in Listing 7.1 takes expected time $3U + 10U = 13U$ (neglecting the time to do the set operation, which assuming is quick); the approach in Listing 7.2 takes expected time $3U + 0.1 * 10U = 4U$ and approach in Listing 7.3 takes expected time $10U + 0.5 * 3U = 11.5U$. Comparing these plans, It can be seen that the second plan is the best, and so is the one chosen for execution.

I thus use an *automatic* predicate ordering approach to provide speed gains, based on offline training. It is noted that predicate ordering works for all remote predicates (the ones requiring CAD modeler computation), and thus concerns not only the lazily evaluated predicates but including the ones in `WHERE` statements. However, as the latter usually do not take much time, the performance gain mainly comes from the lazily evaluated predicates.

Query optimization in database systems includes reordering subtasks in a query to make it more efficient—if a series of filters is applied, the ideal situation is the first filter to reject as much as possible so that subsequent filters have fewer data to process. Standard database query optimization chooses an approach based on statistical information, including the fraction of column entries that are null, the average size of column entries, whether the number of distinct values is likely to increase as the table grows or not, and so on [The15b]. It is usually assumed that retrieving each data item takes a constant amount of time, whereas in the system, some information must be computed by the CAD modeler, and so the time taken may vary considerably according to the predicate involved. I, therefore, modify the standard database query optimizer to take this into account.

My approach is based on the idea of *retention*, the probability that a given predicate will return True. In a `HAVING` clause with multiple predicates, they may be evaluated in any order without affecting the result. If all predicates took the same time to evaluate, for efficiency, they should thus be evaluated in increasing order of retention, to reject as much as possible early on. However, some take longer to evaluate, which should also be taken into account: if all predicates were equally likely to be false, the fastest ones should be evaluated first to reduce the number of slower evaluations. These two requirements can be combined into an overall optimal order of evaluating the predicates by defining *expense*, $E = r * c$, where r is the retention of a predicate, and c is the expected time taken to evaluate it. The fastest way to evaluate a clause is to evaluate the predicates in order of increasing expense. Specifically, the quickest ones and the ones most likely to return False should be evaluated first.

However, in general, neither the retention nor the cost of executing a given predicate for a given model are known. Nevertheless, these quantities can be estimated by a prior offline analysis of a collection of CAD models. Ideally these would be models of a similar kind to the one being considered—a collection of similar water pumps, for example, for finding features in a water pump.

Let $P(a_1, \ldots, a_n)$ be a predicate with n arguments, which for simplicity is taken to be discrete values. Suppose the training set has M models. The retention for the kth model taken individually is

$$r_k = O_k / I_k, \tag{7.1}$$

where O_k is the number of entities (vertices, edges, faces or subfeatures) in model k for which the predicate P is true, and I_k is the number of entities in model k that P can be applied to. The average retention of this predicate over the whole training set is

$$E(r) = \sum_{1}^{M} O_k / \sum_{1}^{M} I_k. \tag{7.2}$$

When predicates involve continuous values, the definition of retention needs to be modified somewhat. For example, face area is a continuous variable, with a corresponding predicate that checks if it is within a given range:
`face_area_in_range(face:int, rmin:real, rmax:real).`
Selectivity is now

$$r = \int_{rmin}^{rmax} P(A)\, dA \tag{7.3}$$

where $P(A)$ is the probability density that an arbitrary face has a certain area. In practice, this may be estimated by constructing a histogram of face areas for all models.

The average time for executing each predicate can also be estimated by processing the same collection of models offline.

Suppose a query has two predicates p_1 and p_2, with average costs c_1 and c_2 and average retentions r_1 and r_2. Suppose there are W data items returned by `WHERE`. The times taken to execute these in different orders can be estimated to be:

$$\begin{aligned} p_1 \text{ then } p_2: \quad & t_{12} = Wc_1 + Wr_1c_2 \qquad (7.4) \\ p_2 \text{ then } p_1: \quad & t_{21} = Wc_2 + Wr_2c_1, \end{aligned}$$

and choose the order of execution accordingly. This analysis may be readily generalised to larger numbers of predicates.

7.2 Experiments

7.2.1 Lazy Evaluation and Caching

Lazy evaluation is expected to be most effective for features with expensive CAD modeler calculations. In this section, I show the effectiveness by comparing the performance of eager and lazy evaluation, with caching or not.

Take a recognition task with a face area constraint, for example. There are five alternative ways below to find feature instances:

1. Eager evaluation (a). Pre-calculate and cache all areas in a local table, and translate the corresponding constraint into a filter predicate in a `WHERE` clause.
2. Eager evaluation (b). Express area computations as remote CAD functions, translate constraints into filter predicates in `WHERE` clauses, and evaluate all area computations at execution time by calling CADfix.
3. Eager evaluation (c). Express area computations as remote CAD functions, and translate constraints into filter predicates in `WHERE` clauses. A local table is used to memoize returned areas so that CADfix is only asked to compute them once.
4. Lazy evaluation (a). Express area computations as remote CAD functions, translate them via `HAVING` clauses, and evaluate all area computations at execution time by calling CADfix.
5. Lazy evaluation (b). Express area computations as remote CAD functions, translate them via `HAVING` clauses. A local table is used to memoize returned areas so that CADfix is only asked to compute them once.

This leads, for example, to the following different ways of finding large through-hole features:

```
CREATE TABLE throughhole AS
SELECT full_edge_e1.edge AS e1, full_edge_e2.edge AS e2, full_edge_e3.edge AS e3,
       full_edge_e4.edge AS e4, full_edge_e5.edge AS e5, full_edge_e6.edge AS e6,
       full_edge_e1.face1 AS f1, full_edge_e3.face1 AS f2, full_edge_e1.face2 AS f3,
       full_edge_e2.face2 AS f4
FROM   full_edge full_edge_e6, full_edge full_edge_e5, full_edge full_edge_e4,
       full_edge full_edge_e3, full_edge full_edge_e2, full_edge full_edge_e1,
       face_area fa, face_area fb
WHERE full_edge_e1.face1=full_edge_e2.face1
  AND full_edge_e1.face2=full_edge_e3.face2
  AND full_edge_e1.face2=full_edge_e5.face1
  AND full_edge_e1.face2=full_edge_e6.face2
  AND full_edge_e2.face2=full_edge_e4.face2
  AND full_edge_e2.face2=full_edge_e5.face2
  AND full_edge_e2.face2=full_edge_e6.face1
  AND full_edge_e3.face1=full_edge_e4.face1
  AND full_edge_e3.face2=full_edge_e5.face1
  AND full_edge_e3.face2=full_edge_e6.face2
  AND full_edge_e4.face2=full_edge_e5.face2
  AND full_edge_e4.face2=full_edge_e6.face1
  AND full_edge_e1.convexity=2 AND full_edge_e2.convexity=2
  AND full_edge_e3.convexity=2 AND full_edge_e4.convexity=2
  AND full_edge_e5.convexity=3 AND full_edge_e6.convexity=3
  AND full_edge_e3.face1<>full_edge_e2.face2
  AND full_edge_e3.face1<>full_edge_e1.face2
  AND full_edge_e3.face1<>full_edge_e1.face1
  AND full_edge_e2.face2<>full_edge_e1.face2
  AND full_edge_e2.face2<>full_edge_e1.face1
  AND full_edge_e1.face2<>full_edge_e1.face1
  AND full_edge_e6.edge<>full_edge_e5.edge
  AND full_edge_e6.edge<>full_edge_e4.edge
  AND full_edge_e6.edge<>full_edge_e3.edge
  AND full_edge_e6.edge<>full_edge_e2.edge
  AND full_edge_e6.edge<>full_edge_e1.edge
  AND full_edge_e5.edge<>full_edge_e4.edge
  AND full_edge_e5.edge<>full_edge_e3.edge
  AND full_edge_e5.edge<>full_edge_e2.edge
  AND full_edge_e5.edge<>full_edge_e1.edge
  AND full_edge_e4.edge<>full_edge_e3.edge
  AND full_edge_e4.edge<>full_edge_e2.edge
  AND full_edge_e4.edge<>full_edge_e1.edge
  AND full_edge_e3.edge<>full_edge_e2.edge
  AND full_edge_e3.edge<>full_edge_e1.edge
  AND full_edge_e2.edge<>full_edge_e1.edge
  AND fa.face=full_edge_e1.face2 AND fb.face=full_edge_e2.face2
  AND fa.area > 100 AND fb.area > 100;
```

Listing 7.4: Eager evaluation (a)

```
CREATE TABLE throughhole AS
SELECT full_edge_e1.edge AS e1, full_edge_e2.edge AS e2, full_edge_e3.edge AS e3,
       full_edge_e4.edge AS e4, full_edge_e5.edge AS e5, full_edge_e6.edge AS e6,
       full_edge_e1.face1 AS f1, full_edge_e3.face1 AS f2, full_edge_e1.face2 AS f3,
       full_edge_e2.face2 AS f4
FROM   full_edge full_edge_e6, full_edge full_edge_e5, full_edge full_edge_e4,
        full_edge full_edge_e3, full_edge full_edge_e2, full_edge full_edge_e1
WHERE full_edge_e1.face1=full_edge_e2.face1
  AND full_edge_e1.face2=full_edge_e3.face2
  AND full_edge_e1.face2=full_edge_e5.face1
  AND full_edge_e1.face2=full_edge_e6.face2
  AND full_edge_e2.face2=full_edge_e4.face2
  AND full_edge_e2.face2=full_edge_e5.face2
  AND full_edge_e2.face2=full_edge_e6.face1
  AND full_edge_e3.face1=full_edge_e4.face1
  AND full_edge_e3.face2=full_edge_e5.face1
  AND full_edge_e3.face2=full_edge_e6.face2
  AND full_edge_e4.face2=full_edge_e5.face2
  AND full_edge_e4.face2=full_edge_e6.face1
  AND full_edge_e1.convexity=2 AND full_edge_e2.convexity=2
  AND full_edge_e3.convexity=2 AND full_edge_e4.convexity=2
  AND full_edge_e5.convexity=3 AND full_edge_e6.convexity=3
  AND full_edge_e3.face1<>full_edge_e2.face2
  AND full_edge_e3.face1<>full_edge_e1.face2
  AND full_edge_e3.face1<>full_edge_e1.face1
  AND full_edge_e2.face2<>full_edge_e1.face2
  AND full_edge_e2.face2<>full_edge_e1.face1
  AND full_edge_e1.face2<>full_edge_e1.face1
  AND full_edge_e6.edge<>full_edge_e5.edge
  AND full_edge_e6.edge<>full_edge_e4.edge
  AND full_edge_e6.edge<>full_edge_e3.edge
  AND full_edge_e6.edge<>full_edge_e2.edge
  AND full_edge_e6.edge<>full_edge_e1.edge
  AND full_edge_e5.edge<>full_edge_e4.edge
  AND full_edge_e5.edge<>full_edge_e3.edge
  AND full_edge_e5.edge<>full_edge_e2.edge
  AND full_edge_e5.edge<>full_edge_e1.edge
  AND full_edge_e4.edge<>full_edge_e3.edge
  AND full_edge_e4.edge<>full_edge_e2.edge
  AND full_edge_e4.edge<>full_edge_e1.edge
  AND full_edge_e3.edge<>full_edge_e2.edge
  AND full_edge_e3.edge<>full_edge_e1.edge
  AND full_edge_e2.edge<>full_edge_e1.edge
  AND calc_area(full_edge_e1.face2)>100
  AND calc_area(full_edge_e2.face2)>100;
```

Listing 7.5: Eager evaluation (b)

```
CREATE TABLE throughhole AS
SELECT full_edge_e1.edge AS e1, full_edge_e2.edge AS e2, full_edge_e3.edge AS e3,
       full_edge_e4.edge AS e4, full_edge_e5.edge AS e5, full_edge_e6.edge AS e6,
       full_edge_e1.face1 AS f1, full_edge_e3.face1 AS f2, full_edge_e1.face2 AS f3,
       full_edge_e2.face2 AS f4
FROM   full_edge full_edge_e6, full_edge full_edge_e5, full_edge full_edge_e4,
       full_edge full_edge_e3, full_edge full_edge_e2, full_edge full_edge_e1
WHERE full_edge_e1.face1=full_edge_e2.face1
  AND full_edge_e1.face2=full_edge_e3.face2
  AND full_edge_e1.face2=full_edge_e5.face1
  AND full_edge_e1.face2=full_edge_e6.face2
  AND full_edge_e2.face2=full_edge_e4.face2
  AND full_edge_e2.face2=full_edge_e5.face2
  AND full_edge_e2.face2=full_edge_e6.face1
  AND full_edge_e3.face1=full_edge_e4.face1
  AND full_edge_e3.face2=full_edge_e5.face1
  AND full_edge_e3.face2=full_edge_e6.face2
  AND full_edge_e4.face2=full_edge_e5.face2
  AND full_edge_e4.face2=full_edge_e6.face1
  AND full_edge_e1.convexity=2 AND full_edge_e2.convexity=2
  AND full_edge_e3.convexity=2 AND full_edge_e4.convexity=2
  AND full_edge_e5.convexity=3 AND full_edge_e6.convexity=3
  AND full_edge_e3.face1<>full_edge_e2.face2
  AND full_edge_e3.face1<>full_edge_e1.face2
  AND full_edge_e3.face1<>full_edge_e1.face1
  AND full_edge_e2.face2<>full_edge_e1.face2
  AND full_edge_e2.face2<>full_edge_e1.face1
  AND full_edge_e1.face2<>full_edge_e1.face1
  AND full_edge_e6.edge<>full_edge_e5.edge
  AND full_edge_e6.edge<>full_edge_e4.edge
  AND full_edge_e6.edge<>full_edge_e3.edge
  AND full_edge_e6.edge<>full_edge_e2.edge
  AND full_edge_e6.edge<>full_edge_e1.edge
  AND full_edge_e5.edge<>full_edge_e4.edge
  AND full_edge_e5.edge<>full_edge_e3.edge
  AND full_edge_e5.edge<>full_edge_e2.edge
  AND full_edge_e5.edge<>full_edge_e1.edge
  AND full_edge_e4.edge<>full_edge_e3.edge
  AND full_edge_e4.edge<>full_edge_e2.edge
  AND full_edge_e4.edge<>full_edge_e1.edge
  AND full_edge_e3.edge<>full_edge_e2.edge
  AND full_edge_e3.edge<>full_edge_e1.edge
  AND full_edge_e2.edge<>full_edge_e1.edge
  AND get_area(full_edge_e1.face2)>100
  AND get_area(full_edge_e2.face2)>100;
```

Listing 7.6: Eager evaluation (c)

In this query, the remote CAD predicate `get_area` is defined as:

```
CREATE OR REPLACE FUNCTION get_area (face integer) RETURNS
    float AS $$  DECLARE  RSLT float;
BEGIN
SELECT area INTO rslt FROM face_area WHERE face_area.face=$1;
        IF NOT FOUND THEN
            rslt:= calc_area ($1);
            INSERT INTO face_area VALUES ($1,rslt);
        END IF;
RETURN rslt;
END;
$$ LANGUAGE plpgsql;
```

Listing 7.7: Function call with memoization

```
CREATE TABLE throughhole AS
SELECT full_edge_e1.edge AS e1, full_edge_e2.edge AS e2, full_edge_e3.edge AS e3,
       full_edge_e4.edge AS e4, full_edge_e5.edge AS e5, full_edge_e6.edge AS e6,
       full_edge_e1.face1 AS f1, full_edge_e3.face1 AS f2, full_edge_e1.face2 AS f3,
       full_edge_e2.face2 AS f4
FROM   full_edge full_edge_e6, full_edge full_edge_e5, full_edge full_edge_e4,
       full_edge full_edge_e3, full_edge full_edge_e2, full_edge full_edge_e1
WHERE full_edge_e1.face1=full_edge_e2.face1
  AND full_edge_e1.face2=full_edge_e3.face2
  AND full_edge_e1.face2=full_edge_e5.face1
  AND full_edge_e1.face2=full_edge_e6.face2
  AND full_edge_e2.face2=full_edge_e4.face2
  AND full_edge_e2.face2=full_edge_e5.face2
  AND full_edge_e2.face2=full_edge_e6.face1
  AND full_edge_e3.face1=full_edge_e4.face1
  AND full_edge_e3.face2=full_edge_e5.face1
  AND full_edge_e3.face2=full_edge_e6.face2
  AND full_edge_e4.face2=full_edge_e5.face2
  AND full_edge_e4.face2=full_edge_e6.face1
  AND full_edge_e1.convexity=2 AND full_edge_e2.convexity=2
  AND full_edge_e3.convexity=2 AND full_edge_e4.convexity=2
  AND full_edge_e5.convexity=3 AND full_edge_e6.convexity=3
  AND full_edge_e3.face1<>full_edge_e2.face2
  AND full_edge_e3.face1<>full_edge_e1.face2
  AND full_edge_e3.face1<>full_edge_e1.face1
  AND full_edge_e2.face2<>full_edge_e1.face2
  AND full_edge_e2.face2<>full_edge_e1.face1
  AND full_edge_e1.face2<>full_edge_e1.face1
  AND full_edge_e6.edge<>full_edge_e5.edge
  AND full_edge_e6.edge<>full_edge_e4.edge
  AND full_edge_e6.edge<>full_edge_e3.edge
  AND full_edge_e6.edge<>full_edge_e2.edge
  AND full_edge_e6.edge<>full_edge_e1.edge
  AND full_edge_e5.edge<>full_edge_e4.edge
  AND full_edge_e5.edge<>full_edge_e3.edge
  AND full_edge_e5.edge<>full_edge_e2.edge
  AND full_edge_e5.edge<>full_edge_e1.edge
  AND full_edge_e4.edge<>full_edge_e3.edge
  AND full_edge_e4.edge<>full_edge_e2.edge
  AND full_edge_e4.edge<>full_edge_e1.edge
  AND full_edge_e3.edge<>full_edge_e2.edge
  AND full_edge_e3.edge<>full_edge_e1.edge
  AND full_edge_e2.edge<>full_edge_e1.edge
GROUP BY full_edge_e1.edge, full_edge_e2.edge, full_edge_e3.edge,
         full_edge_e4.edge, full_edge_e5.edge, full_edge_e6.edge,
         full_edge_e1.face1, full_edge_e3.face1, full_edge_e1.face2,
         full_edge_e2.face2
HAVING calc_area(full_edge_e1.face2)>100 AND calc_area(full_edge_e2.face2)>100;
```

Listing 7.8: Lazy evaluation (a)

```
CREATE TABLE throughhole AS
SELECT full_edge_e1.edge AS e1, full_edge_e2.edge AS e2, full_edge_e3.edge AS e3,
       full_edge_e4.edge AS e4, full_edge_e5.edge AS e5, full_edge_e6.edge AS e6,
       full_edge_e1.face1 AS f1, full_edge_e3.face1 AS f2, full_edge_e1.face2 AS f3,
       full_edge_e2.face2 AS f4
FROM   full_edge full_edge_e6, full_edge full_edge_e5, full_edge full_edge_e4,
       full_edge full_edge_e3, full_edge full_edge_e2, full_edge full_edge_e1
WHERE full_edge_e1.face1=full_edge_e2.face1
  AND full_edge_e1.face2=full_edge_e3.face2
  AND full_edge_e1.face2=full_edge_e5.face1
  AND full_edge_e1.face2=full_edge_e6.face2
  AND full_edge_e2.face2=full_edge_e4.face2
  AND full_edge_e2.face2=full_edge_e5.face2
  AND full_edge_e2.face2=full_edge_e6.face1
  AND full_edge_e3.face1=full_edge_e4.face1
  AND full_edge_e3.face2=full_edge_e5.face1
  AND full_edge_e3.face2=full_edge_e6.face2
  AND full_edge_e4.face2=full_edge_e5.face2
  AND full_edge_e4.face2=full_edge_e6.face1
  AND full_edge_e1.convexity=2 AND full_edge_e2.convexity=2
  AND full_edge_e3.convexity=2 AND full_edge_e4.convexity=2
  AND full_edge_e5.convexity=3 AND full_edge_e6.convexity=3
  AND full_edge_e3.face1<>full_edge_e2.face2
  AND full_edge_e3.face1<>full_edge_e1.face2
  AND full_edge_e3.face1<>full_edge_e1.face1
  AND full_edge_e2.face2<>full_edge_e1.face2
  AND full_edge_e2.face2<>full_edge_e1.face1
  AND full_edge_e1.face2<>full_edge_e1.face1
  AND full_edge_e6.edge<>full_edge_e5.edge
  AND full_edge_e6.edge<>full_edge_e4.edge
  AND full_edge_e6.edge<>full_edge_e3.edge
  AND full_edge_e6.edge<>full_edge_e2.edge
  AND full_edge_e6.edge<>full_edge_e1.edge
  AND full_edge_e5.edge<>full_edge_e4.edge
  AND full_edge_e5.edge<>full_edge_e3.edge
  AND full_edge_e5.edge<>full_edge_e2.edge
  AND full_edge_e5.edge<>full_edge_e1.edge
  AND full_edge_e4.edge<>full_edge_e3.edge
  AND full_edge_e4.edge<>full_edge_e2.edge
  AND full_edge_e4.edge<>full_edge_e1.edge
  AND full_edge_e3.edge<>full_edge_e2.edge
  AND full_edge_e3.edge<>full_edge_e1.edge
  AND full_edge_e2.edge<>full_edge_e1.edge
GROUP BY full_edge_e1.edge, full_edge_e2.edge, full_edge_e3.edge,
         full_edge_e4.edge, full_edge_e5.edge, full_edge_e6.edge,
         full_edge_e1.face1, full_edge_e3.face1, full_edge_e1.face2,
         full_edge_e2.face2
HAVING get_area(full_edge_e1.face2) > 100 AND get_area(full_edge_e2.face2)>100;
```

Listing 7.9: Lazy evaluation (b)

The following three experiments are performed to determine the impact of lazy evaluation; Table 7.4 gives the numbers of features found and Table 7.5 gives the times taken to find these features in the reducer model in Fig. 6.12.

Task 1. Find open slots with side face area greater than 20 units, and bottom face area greater than 2 units;

Task 2. Find all through-holes with side face area less than 550 units and bore area smaller than 50 units;

Task 3. Find all through-holes with cylindrical faces area greater than 100 units;

Task	Task 1	Task 2	Task 3
Number of features, any size	140	164	164
Number of features, specified size	35	18	142

Table 7.4: Feature finding results for various tasks

As Table 7.5 shows, lazy evaluation approach (b) achieved the best performance in each case, being about 6 to 8 times faster than eager evaluation approach (a), and much better than eager evaluation approaches (b) and (c). It is also about twice as fast as lazy evaluation approach (a). Using eager evaluation (a) takes about the same time for each task, because of the similar procedure—first calculate areas of all faces, memoize them in a local table and then perform a filter-based query; time is dominated by the area calculations. Eager evaluation (b) is slowest: the area of each face is evaluated multiple times. Eager evaluation (c) is better, as caching means that areas are only computed

Experiment	Task 1	Task 2	Task 3
Eager evaluation (a)	1216	1323	1314
Eager evaluation (b)	40311	51286	53309
Eager evaluation (c)	11399	15471	144262
Lazy evaluation (a)	228	1279	401
Lazy evaluation (b)	163	210	156

Table 7.5: Feature finding times (in milliseconds) for the reducer models, using different evaluation strategies, and different tasks.

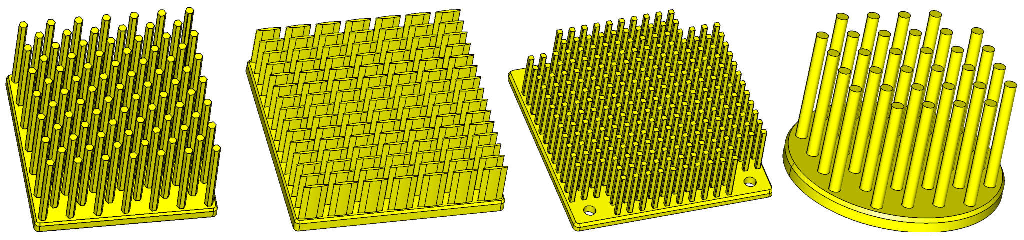


Figure 7.1: Examples from CPU heatsink training set

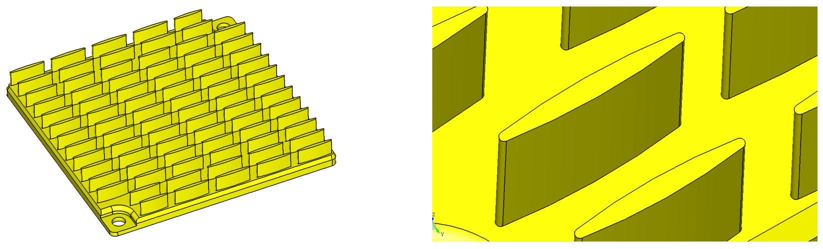


Figure 7.2: Model for finding small fins, and detail

once. For the same reason, lazy evaluation approach (b) outperforms lazy evaluation approach (a).

7.2.2 Predicate Ordering

More complicated features may be defined by multiple expensive attribute predicates. In this section, I show further experiments that not only use lazy evaluation but also predicate ordering.

Determining average times and retention requires offline training on a large model set. For this, I use 826 real industrial models of CPU heat sinks downloaded from [Ltd15]. Examples of these models are shown in Fig. 7.1.

The model in which features are to be found is shown in Fig. 7.2, which, like other CPU heat sinks, includes a large base and fins of several different sizes. Each fin is composed of two cylindrical faces with two tangentially connected side faces, a top face and a bottom edge loop.

Predicate	average execution time	average retention
area_in_range	1 ms	0.2231
perimeter_in_range	0.003 ms	0.0084

Table 7.6: Retention and cost of target predicates

The feature recognition task here is to find *small fins*, defined as fins whose side face area is between 10 and 20 square units, and whose top face has perimeter between 18.5 and 18.7 units. This requires the predicates

```
area_in_range(full_edge_e4.face2, 10, 20)
perimeter_in_range(full_edge_e1.face1, 18.5, 18.7);
```

In the feature finder, these range predicates are translated to enable PostgreSQL to use lazy evaluation with memoization; the function `calc_area` is a remote CAD function call to CADfix.

```
CREATE OR REPLACE FUNCTION area_in_range(face int, lv float, hv float) RETURNS boolean
     AS
$$ DECLARE
   RSLT float; VAL boolean;
BEGIN
SELECT area INTO rslt
FROM face_area
WHERE face_area.face=$1;
IF NOT FOUND
   THEN rslt:= calc_area ($1);
   INSERT INTO face_area VALUES ($1,rslt);
END IF;
RETURN ((rslt > $2) and (rslt < $3)) ;
END;
$$ LANGUAGE plpgsql;
```

To estimate the cost and retention of the area and perimeter functions, offline training was performed on the set of training models. The area and perimeter distributions are shown in Fig. 7.3. Average times to compute these properties, and their retention for the particular ranges of values used in the test, are given in Table 7.6.

The target query used to find small fins is as follows, where the two predicates in the

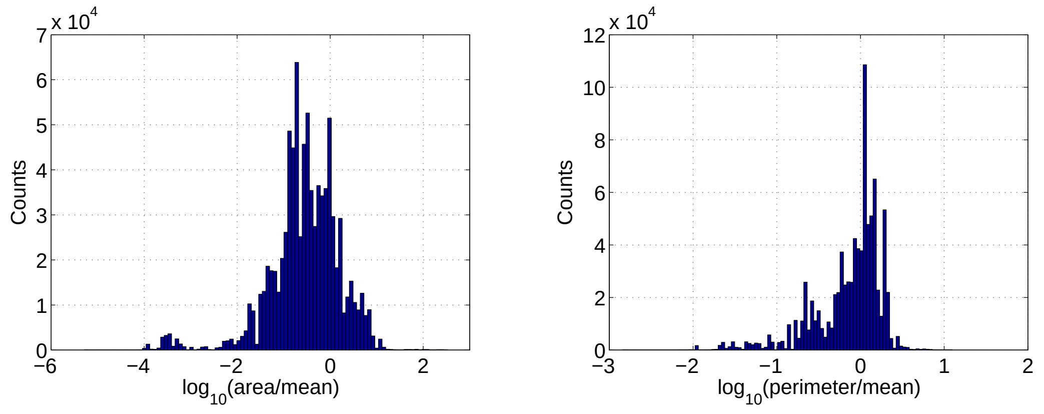


Figure 7.3: Histograms of face areas and perimeters for training model dataset

`HAVING` clause are the ones being considered for reordering:

```
SELECT full_edge_e1.edge AS e1, full_edge_e2.edge AS e2, full_edge_e3.edge AS e3,
       full_edge_e4.edge AS e4, full_edge_e5.edge AS e5, full_edge_e6.edge AS e6,
       full_edge_e7.edge AS e7, full_edge_e8.edge AS e8, full_edge_e9.edge AS e9,
       full_edge_e10.edge AS e10, full_edge_e11.edge AS e11,
       full_edge_e12.edge AS e12, full_edge_e1.face1 AS f1,
       full_edge_e1.face2 AS f2, full_edge_e4.face2 AS f3,
       full_edge_e3.face2 AS f4, full_edge_e2.face2 AS f5,
       full_edge_e9.face2 AS f6
FROM full_edge full_edge_e12, full_edge full_edge_e11, full_edge full_edge_e10,
     full_edge full_edge_e9, full_edge full_edge_e8, full_edge full_edge_e7,
     full_edge full_edge_e6, full_edge full_edge_e5, full_edge full_edge_e4,
     full_edge full_edge_e3, full_edge full_edge_e2, full_edge full_edge_e1
WHERE full_edge_e1.face1=full_edge_e2.face1
  AND full_edge_e1.face1=full_edge_e3.face1
  AND full_edge_e1.face1=full_edge_e4.face1
  AND full_edge_e1.face2=full_edge_e5.face1
  AND full_edge_e1.face2=full_edge_e6.face1
  AND full_edge_e1.face2=full_edge_e9.face1
  AND full_edge_e2.face1=full_edge_e3.face1
  AND full_edge_e2.face1=full_edge_e4.face1
  AND full_edge_e2.face2=full_edge_e5.face2
  AND full_edge_e2.face2=full_edge_e8.face2
  AND full_edge_e2.face2=full_edge_e12.face2
  AND full_edge_e3.face1=full_edge_e4.face1
  AND full_edge_e3.face2=full_edge_e7.face2
  AND full_edge_e3.face2=full_edge_e8.face1
  AND full_edge_e3.face2=full_edge_e11.face1
  AND full_edge_e4.face2=full_edge_e6.face2
  AND full_edge_e4.face2=full_edge_e7.face1
  AND full_edge_e4.face2=full_edge_e10.face2
  AND full_edge_e5.face1=full_edge_e6.face1
  AND full_edge_e5.face2=full_edge_e8.face2
  AND full_edge_e5.face1=full_edge_e9.face1
  AND full_edge_e5.face2=full_edge_e12.face2
  AND full_edge_e6.face2=full_edge_e7.face1
  AND full_edge_e6.face1=full_edge_e9.face1
  AND full_edge_e6.face2=full_edge_e10.face2
  AND full_edge_e7.face2=full_edge_e8.face1
  AND full_edge_e7.face1=full_edge_e10.face2
  AND full_edge_e7.face2=full_edge_e11.face1
  AND full_edge_e8.face1=full_edge_e11.face1
  AND full_edge_e8.face2=full_edge_e12.face2
  AND full_edge_e9.face2=full_edge_e10.face1
  AND full_edge_e9.face2=full_edge_e11.face2
  AND full_edge_e9.face2=full_edge_e12.face1
  AND full_edge_e10.face1=full_edge_e11.face2
  AND full_edge_e10.face1=full_edge_e12.face1
  AND full_edge_e11.face2=full_edge_e12.face1
  AND full_edge_e1.convexity=2 AND full_edge_e2.convexity=2
  AND full_edge_e3.convexity=2 AND full_edge_e4.convexity=2
  AND full_edge_e5.convexity=3 AND full_edge_e6.convexity=3
  AND full_edge_e7.convexity=3 AND full_edge_e8.convexity=3
  AND full_edge_e9.convexity=1 AND full_edge_e10.convexity=1
  AND full_edge_e11.convexity=1 AND full_edge_e12.convexity=1
  AND full_edge_e12.edge<>full_edge_e11.edge
  AND full_edge_e12.edge<>full_edge_e10.edge
  AND full_edge_e12.edge<>full_edge_e9.edge
  AND full_edge_e12.edge<>full_edge_e8.edge
  AND full_edge_e12.edge<>full_edge_e7.edge
  AND full_edge_e12.edge<>full_edge_e6.edge
  AND full_edge_e12.edge<>full_edge_e5.edge
  AND full_edge_e12.edge<>full_edge_e4.edge
  AND full_edge_e12.edge<>full_edge_e3.edge
  AND full_edge_e12.edge<>full_edge_e2.edge
  AND full_edge_e12.edge<>full_edge_e1.edge
  AND full_edge_e11.edge<>full_edge_e10.edge
  AND full_edge_e11.edge<>full_edge_e9.edge
  AND full_edge_e11.edge<>full_edge_e8.edge
  AND full_edge_e11.edge<>full_edge_e7.edge
  AND full_edge_e11.edge<>full_edge_e6.edge
  AND full_edge_e11.edge<>full_edge_e5.edge
  AND full_edge_e11.edge<>full_edge_e4.edge
  AND full_edge_e11.edge<>full_edge_e3.edge
  AND full_edge_e11.edge<>full_edge_e2.edge
  AND full_edge_e11.edge<>full_edge_e1.edge
  AND full_edge_e10.edge<>full_edge_e9.edge
  AND full_edge_e10.edge<>full_edge_e8.edge
  AND full_edge_e10.edge<>full_edge_e7.edge
  AND full_edge_e10.edge<>full_edge_e6.edge
  AND full_edge_e10.edge<>full_edge_e5.edge
  AND full_edge_e10.edge<>full_edge_e4.edge
  AND full_edge_e10.edge<>full_edge_e3.edge
  AND full_edge_e10.edge<>full_edge_e2.edge
  AND full_edge_e10.edge<>full_edge_e1.edge
  AND full_edge_e9.edge<>full_edge_e8.edge
  AND full_edge_e9.edge<>full_edge_e7.edge
  AND full_edge_e9.edge<>full_edge_e6.edge
  AND full_edge_e9.edge<>full_edge_e5.edge
  AND full_edge_e9.edge<>full_edge_e4.edge
  AND full_edge_e9.edge<>full_edge_e3.edge
  AND full_edge_e9.edge<>full_edge_e2.edge
  AND full_edge_e9.edge<>full_edge_e1.edge
  AND full_edge_e8.edge<>full_edge_e7.edge
  AND full_edge_e8.edge<>full_edge_e6.edge
  AND full_edge_e8.edge<>full_edge_e5.edge
  AND full_edge_e8.edge<>full_edge_e4.edge
  AND full_edge_e8.edge<>full_edge_e3.edge
  AND full_edge_e8.edge<>full_edge_e2.edge
  AND full_edge_e8.edge<>full_edge_e1.edge
  AND full_edge_e7.edge<>full_edge_e6.edge
  AND full_edge_e7.edge<>full_edge_e5.edge
  AND full_edge_e7.edge<>full_edge_e4.edge
  AND full_edge_e7.edge<>full_edge_e3.edge
  AND full_edge_e7.edge<>full_edge_e2.edge
  AND full_edge_e7.edge<>full_edge_e1.edge
  AND full_edge_e6.edge<>full_edge_e5.edge
  AND full_edge_e6.edge<>full_edge_e4.edge
  AND full_edge_e6.edge<>full_edge_e3.edge
  AND full_edge_e6.edge<>full_edge_e2.edge
  AND full_edge_e6.edge<>full_edge_e1.edge
  AND full_edge_e5.edge<>full_edge_e4.edge
  AND full_edge_e5.edge<>full_edge_e3.edge
  AND full_edge_e5.edge<>full_edge_e2.edge
  AND full_edge_e5.edge<>full_edge_e1.edge
  AND full_edge_e4.edge<>full_edge_e3.edge
  AND full_edge_e4.edge<>full_edge_e2.edge
```

```
  AND full_edge_e4.edge<>full_edge_e1.edge
  AND full_edge_e3.edge<>full_edge_e2.edge
  AND full_edge_e3.edge<>full_edge_e1.edge
  AND full_edge_e2.edge<>full_edge_e1.edge
  AND face_geometry_is(full_edge_e1.face2,2006)
  AND face_geometry_is(full_edge_e3.face2,2006)
GROUP BY full_edge_e1.edge, full_edge_e2.edge, full_edge_e3.edge,
       full_edge_e4.edge, full_edge_e5.edge, full_edge_e6.edge,
       full_edge_e7.edge, full_edge_e8.edge, full_edge_e9.edge,
       full_edge_e10.edge, full_edge_e11.edge, full_edge_e12.edge,
       full_edge_e1.face1, full_edge_e1.face2, full_edge_e4.face2,
       full_edge_e3.face2, full_edge_e2.face2, full_edge_e9.face2
HAVING area_in_range(full_edge_e4.face2, 10, 20)
AND perimeter_in_range(full_edge_e1.face1, 18.5, 18.7);
```

Listing 7.10: Query to find small fins

Results obtained using the system based on the PostgreSQL engine with lazy evaluation, with and without predicate reordering are compared. The test was repeated 100 times to give an averaged performance result. Each time the PostgreSQL server was restarted, warmed up and the OS caches (pagecache, dentries and inodes) were cleared.

For both versions, the SQL query with the predicates given in either possible order are both timed: area then perimeter, or perimeter then area. With reordering, the query planner always chooses the predicate ordering perimeter then area, whichever ordering the predicates are initially provided in: the much higher cost of computing areas compared to perimeters far outweighs the differences in retention. Without reordering, predicates are simply executed in the sequence given.

Fig. 7.4 give the times taken to find features in each of the 100 runs in each case, using the different strategies. Without reordering, the approaches take different times according to which predicate is evaluated first. Computing area first, most runs take 350–400 ms, while if the perimeter is computed first, most runs take 490–530 ms. However, if reordering is used, no matter how the predicates are ordered in the original definition, the times taken in both cases have closely similar ranges and distributions. Average times are given in Table 7.7. As expected, the optimizer-chosen perimeter-then-area

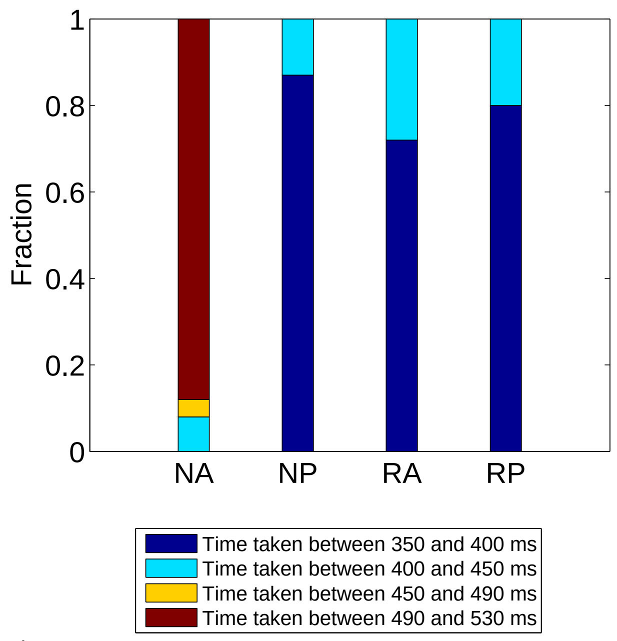


Figure 7.4: Times taken for for each of 100 runs. N = No reordering of predicates. R = Reordering of predicates. A = Area first. P = Perimeter first. .

Method	Area then perimeter	Perimeter then area
No predicate reordering	498 ms	393 ms
With predicate reordering	392 ms	392 ms

Table 7.7: Average feature finding times with and without predicate reordering optimization.

ordering is faster than the alternative area-then-perimeter ordering. The overhead required to perform the retention calculation is negligible.

As an effective translation and lazy evaluation already greatly improve performance, predicate ordering only makes a worthwhile difference for large models. Most of the time spent is running the query itself, rather than the CAD computations, so the saving is not much (21% in this case). Nevertheless, for other more complex predicates that the CAD modeler takes longer to process, the savings could be greater.

In summary, the best approach for a feature recognition which involves numerical attribute predicates is

1. lazily evaluate these attributes to reduce the workload performed by the CAD modeler,
2. memoize the results of function calls, to avoid repeatedly making the same call to the CAD modeler;
3. order the function calls according to execution cost and retention, to minimize the workload performed on the CAD modeler.

7.3 Summary

The approach is extended to incorporate lazy evaluation with memoization and predicate ordering optimizations in this chapter. Lazy evaluation essentially requires query rewriting at the declarative level while predicate ordering is a transformation in algebraic space using statistical information. The optimizations are most effective for (multiple) expensive predicates and do not have much effect for basic features (defined only using bounds and edge convexity predicates). Instead of time complexity improvements, both give a constant factor improvement. Clearly, the cost of the predicate affects the effectiveness of lazy evaluation, and the similarity between the test model and the training set is likely to determine the effectiveness of predicate ordering.

Chapter 8

Discussion

8.1 Conclusion

The thesis has presented an approach to quickly find features in CAD models using DB query optimizations. Firstly, I summary the works addressed in this thesis.

Declarative feature definitions when naively translated into the procedural code leads to nested *for-all* loops to exhaustively find entities satisfying requirements. Such an algorithm is too expensive for real use. When relational DB systems answer queries, they have much in common with a declarative feature recognition task: (i) they use a declarative language (SQL) to express the query and (ii) the query performs an exhaustive search for all tuples in tables. Much research has gone into query optimization to ensure that DB kernels return results quickly. I have demonstrated that if building a testbed around a DB kernel and a CAD modeler, it is able to use query optimization to find features quickly.

Initially, an SQLite based testbed is built to verify the idea. The feature recognizer consists of a translator (to turn a feature definition into an SQL query), an SQLite query optimizer (to turn the SQL query into an algorithm) and a CAD modeler (to read in the model and pass the necessary data to tables). I proposed a straightforward translation in which all predicates (excluding rank predicates such as `Lower_id(e1,e2)`) in the feature definition are turned into *existence test* clauses. Experiments show that this approach can find features in approximately $O(n^2)$ time; examining the query execution plan shows that SQLite uses index access optimizations to achieve this.

I next replaced the database engine with PostgreSQL to see whether the optimizations provided by SQLite could be replicated, and to determine whether different database engines would arrive at similar query execution plans when used for feature recognition. However, the performance observed on SQLite was not repeatable with PostgreSQL, as it does not have automatic indexing optimization. I thus developed a new translation approach in which, instead of using *existence test* subqueries, predicates are first transformed into an internal form and then turned into *access predicates* or *filter predicates*. Experiments show that (i) the resulting algorithm generated by PostgreSQL has $O(n)$ time complexity, while (ii) the algorithm generated by SQLite has $O(n \log(n))$ complexity. Both achieve much better performance than the original SQLite testbed. Execution plans show that PostgreSQL turns the access predicates into a hash join algorithm which can reduce the search space greatly, and on the other hand SQLite uses the access predicates as index keys, allowing it to access tuples of interest quickly. Although they choose different algorithms, both perform very well. It is noted that such access predicates and filter predicates can be optimized by all mainstream DB systems using indexing or join algorithms.

During translation, attribute predicates are turned into functions executed by the CAD modeler and return True or False; in some cases they may be expensive to compute. Performing such a calculation on all entities is likely to be very slow. Further experiments showed that lazy evaluation and predicate ordering are effective techniques for reducing the workload for expensive predicates.

The current approach has two prerequisites: it requires a manifold model, and features to be connected, both of which are likely to be satisfied for most real world problems. Nevertheless, I have also discussed how these limitations may be overcome.

The state-of-the-art performance for feature recognition has been improved, and the first linear complexity feature finder has been demonstrated. Nevertheless, some limitations remain. It is hard to write definitions for complicated features, topological symmetries can result in the same feature being reported multiple times, and the method works no better than previous methods in the presence of interacting features. I hope to consider these problems in future.

In summary, the hypothesis are verified and the aims proposed in section 1.3 are achieved. Specifically:

1. Declarative feature definition language syntax and predicates are defined. End users can describe what he/she want using the declarations; expressive power of the domain specific language is discussed;
2. Various translation rules that turn feature definitions into SQL queries in a general way are discussed;
3. Testbeds based on SQLite and PostgreSQL are devised to verify the hypothesis, linear performance feature recognition is achieved;
4. The execution plans behind the performances are investigated, based on the understanding, a stand alone feature recognizer without using the database systems are proposed in the following subsections.
5. Beyond the database built-in query optimizations, Lazy evaluation with caching and predicate ordering are investigated how to further improve performance for features that are defined using both topological and graphical constraints.

8.2 Contribution

The novel contributions of this work are as follows:

- Definition and validation of architecture for SQL driven feature recognition. It is demonstrate that database query optimization technologies can provide fast feature recognition if using a translator that turns a feature definition into an SQL query, with a CAD modeler providing the model data to it.

- Automatic declarative feature representation to SQL translator. It is investigated that how a *general* translator can be achieved using *existence test* subqueries, *access predicates* or *filter predicates*.
- It is demonstrated that *linear* performance for common feature recognition can be achieved using access predicates based translator and PostgreSQL optimization engine based testbed. This outperforms the prior state-of-the-art.
- It is demonstrated that beyond database system build-in optimizations, further performance improvement can be achieved by 1). *lazy evaluation and caching* to reduce the work performed by the CAD modeler. 2). Estimates of the time taken to compute various geometric operations can be used to further improve the query plan by *reordering filtering operations*.
- It is demonstrated that the approach can find features in complex real world engineering objects in acceptable time.

8.3 Limitations and Future Work

Although the proposed approach shows great promise, there are several limitations and issues that warrant further investigation.

Firstly, the PostgreSQL based approach described in Chapter 6 is targeted at features with connected entities, such as the faces which make up a pocket. It requires features to be defined using `Bounds_EF` or `Bounds_VE`. Nevertheless,

- recognizing such features is the major feature recognition task in CAM, CAPP and CAE;

- using a full-edge data model, `Bounds_EF` and `Bounds_VE` can be turned into access predicates in SQL queries, which in turn are readily-optimized by mainstream DB systems.

In practice, features are not always defined using relational predicates and may not be connected features, e.g. a parallel-faces feature as defined in Listing 8.1: almost-parallel faces could be suitable for a robot to grip an object; parallel faces may be required for symmetry analysis in CAE. The definition includes no relation predicates, only one determining the angle between two face normals.

```
    DEFINE parallel-faces AS
        face: f1, f2
    SATISFYING
        Face_has_geometry(f1, plane)
        Face_has_geometry(f2, plane)
        Plane_normal_angle_between(f1, f2, 0.01)
    END
```

Listing 8.1: Parallel-faces feature

In order to recognize general features, the system can support using primitives as range tables, as does the SQLite based testbed. DB systems can efficiently turn the `full_edge` table into `face` and `edge` tables. In this case, the feature definition can be translated into the naive nested for-all loop algorithm, e.g.,

```
for each f1 in face do
    for each f2 in face do
        if Face_has_geometry(f1, plane) &&
        Face_has_geometry(f1, plane) && (Plane_normal_angle_between(f1,f2) < 0.01)
            add to results (f1, f2)
```

Listing 8.2: Algorithm for parallel-faces feature

As the computation (`Plane_normal_angle_between`) is performed on every pair of faces, it is an exhaustive method. As time is mainly spent on function execution, not on the nested for all loop, thus, the only way to improve performance is to avoid computing the function for pairs of faces that have significantly different orientations.

It is well known that spatial indexes are useful in solid modeling [AK91, Tan92, SM95], and can help to reduce the complexity of such nested loops. For example, an octree can be used to answer spatial queries such as find the nearest object [BR09a, BR09b]. I intend to explore such indexes and other tools such as medial objects [PSB95] for such purposes.

Secondly, the form of declarations the testbed supported do not currently permit features with *variable* numbers of elements, such as a ring of holes, a gear with a variable number of teeth, or a row of slots. These are most easily defined recursively. Some variants of SQL also allow recursive queries, which indicates a potential way of generalizing the method.

Thirdly, automatically identifying topological symmetry, and preventing the same feature being returned multiple times is also a tricky issue. For example, Table 6.1 is the topological symmetry results for a notch feature. Manually defining features (see Listing 3.4) using comparison relation predicates like `Greater_id(id1, id2)` can be tedious and error-prone. Adding clauses automatically to remove all symmetries without also throwing away some desired results is difficult, yet writing correct declarations to do so manually is also difficult. A simple approach, but one that is almost certainly suboptimal, is instead to ignore topological symmetry in the feature definition, and to check for and delete additional copies of the same feature in the returned results. Methods are needed which do not have excessive complexity as the number of entities in a feature, or the number of multiple copies, grows.

Fourthly, use of the full-edge form assumes that the models are manifold. However, it seems plausible that the approach could be generalised to work for non-manifold models, as there will in general only be a small number of faces around each edge, and few non-manifold edges. Alternatively, there is some mature work to change non-manifold models into manifold ones [MHS01, McM00].

Fifthly, different modelers use different internal representations. For example, one modeler may use multiple cylindrical faces to represent a complete cylindrical surface, while another modeler may use just one. Unfortunately, this means that the engineer writing a feature definition must understand the internal representation used by the particular modeler a definition is intended for: feature definitions are unlikely to be interchangeable between modelers. This is, of course, a problem for any feature finder which works on a boundary representation. While merging such cylindrical surfaces may be a useful step in overcoming this particular problem, the general problem is probably as hard as the one of translating CAD data between different systems, which is notoriously tricky [KSL97, RHJ$^+$01, MB08].

Writing declarative definitions can still be a complex task for end users, even if less difficult than devising algorithms. Complicated features may have tens of faces and edges, and it is not easy to label them, write, and debug a correct and complete declaration. A better approach based on a point-and-click interface may help to alleviate this burden, and I intend to investigate it in future.

Finding complex features in the presence of *interacting features* perhaps remains the outstanding problem in feature recognition. It is far from clear that even an assisted declarative approach will let engineers do this effectively—it may be just too hard to take into account all possible interactions. This will only become clear if and when engineers start using such a declarative approach in practice; industrial feedback is needed to clarify this issue. As always, dealing with interacting features is difficult, and the current work offers no clear way to help in this area. More work is needed to understand how to define features in a way that takes interactions into account, and report instances of interacting features in a way that is useful to the user. Possible solutions may require incorporating hint based predicates to define features. Classic works on hint based automatic feature recognition [HR97, GS98] may provide knowledge basement in the declarative feature recognizer.

8.4 A Feature Recognition Architecture

The experiments with DB engines gave good performance in Chapters 5 and 6, and the corresponding execution plans have also let us understand how that good performance was obtained. As a result, a feature recogniser to directly use these methods without any longer needing the database engine can be implemented. The DB engines were a tool for understanding, and not a necessary component of a fast feature recognition system. In this section, I propose a stand-alone feature recognizer architecture, which

1. can recognize general features, both connected features and disconnected features, and
2. does not need a DB kernel, just the knowledge gained from the two testbeds. DB systems usually include hundreds of thousands or millions of lines of source code, many of which are irrelevant in a feature recognizer. A standalone feature recognizer is more concise and may provide better performance.

The standalone declarative feature recognizer includes two main modules: a language compiler and an executor. The compiler turns a feature declaration into several algorithms and the executor executes them.

The workflow of the language compiler is illustrated in Fig. 8.1. The primitive classifier reads in all primitives in the feature definition, compares them with a lookup table and classifies them as type R or type Q (I omit the subfeature here as their translation are rather straightforward). Type R refers to the primitives referred by relational predicates; while type Q are others. In practice, a feature can be defined using entities of only type R, or of type Q, or both. Relational predicates only take effect on type R while attribute predicates can work on both entity types.

I classify the primitives into the two types because the associated predicates use different data models. For example, if features are defined using `Bounds_EF`, the system needs to pre-load $full\_edge(e, f_a, f_b, convexity)$ relations (similarly for `Bounds_VE`)

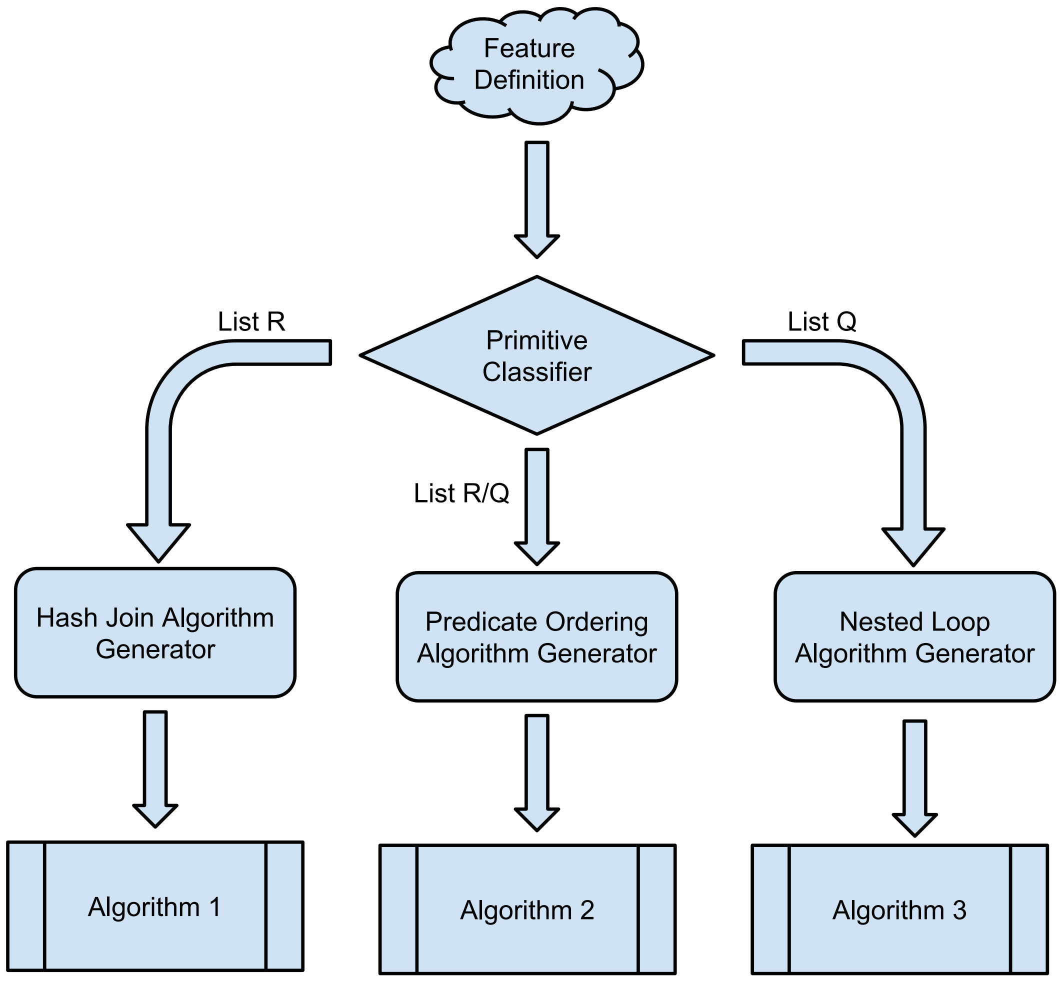

Feature Definition
List R
Primitive Classifier
List Q
List R/Q
Hash Join Algorithm Generator
Predicate Ordering Algorithm Generator
Nested Loop Algorithm Generator
Algorithm 1
Algorithm 2
Algorithm 3

Figure 8.1: Workflow for language compiler

so that hash join algorithm can be used to translate the feature definition. If the feature is defined without type R, the data models should be single column primitive tables, for example $edges(e)$, and a naive nested loop algorithm must be used to translate the feature definition, although in future a spatial index may be a better alternative.

Next, different algorithm generators use the corresponding primitives and predicates to generate procedural algorithms. Specifically:

1. A hash join algorithm generator uses list R and relation predicates (`Bounds_EF` and `Bounds_VE`) to generate a hash join algorithm. This approach corresponds to the optimization described in Chapter 6. The key idea is that the hash join

algorithm reduces the search space which satisfies the access predicates. Pseudocode is given in Listing 2.3.

2. A predicate ordering algorithm generator uses lists R, Q and all attribute predicates to generate an algorithm in which all numerical attribute predicates are turned into functions which call the CAD modeler to do the time-consuming calculations. More importantly, the function calls are ordered so that the overall cost of CAD modeler work is minimized. This approach corresponds to the predicate ordering optimization described in Chapter 7.

3. A nested loop algorithm generator uses list Q to generate naive for-all nested loop algorithms. This algorithm is mainly responsible for non-local parts of definitions, as in the parallel-face feature. Gibson's optimizations [GIS97, GISH97, GIS99] can be used to optimize the nested loops; they can also be speed up by spatial indexing as discussed earlier.

The workflow of the executor is shown in Fig. 8.2. All candidate primitives and predicates in the feature definition are filtered by Algorithms 1 and 3, generating a much smaller candidate set. The smaller set of data is further processed by the functions which are ordered by Algorithm 2. Execution uses lazy evaluation as described in Chapter 7.

next, I briefly analyze the complexity of this approach. In practice, the most likely scenario is to recognize local features, defined using type R primitives and `Bounds_EF` or `Bounds_VE`. The extreme case is to recognize non-local features, e.g. parallel-faces which are defined using type Q primitives. In between, some features have both R and Q. The first case has linear performance as explained in Chapter 6, while the second case has the highest complexity as it is a nested loop algorithm. For the most common third case, when applying Algorithms 1 and 3 in order, the total complexity is $O(n^l)$ where l is the number of primitives in list Q.

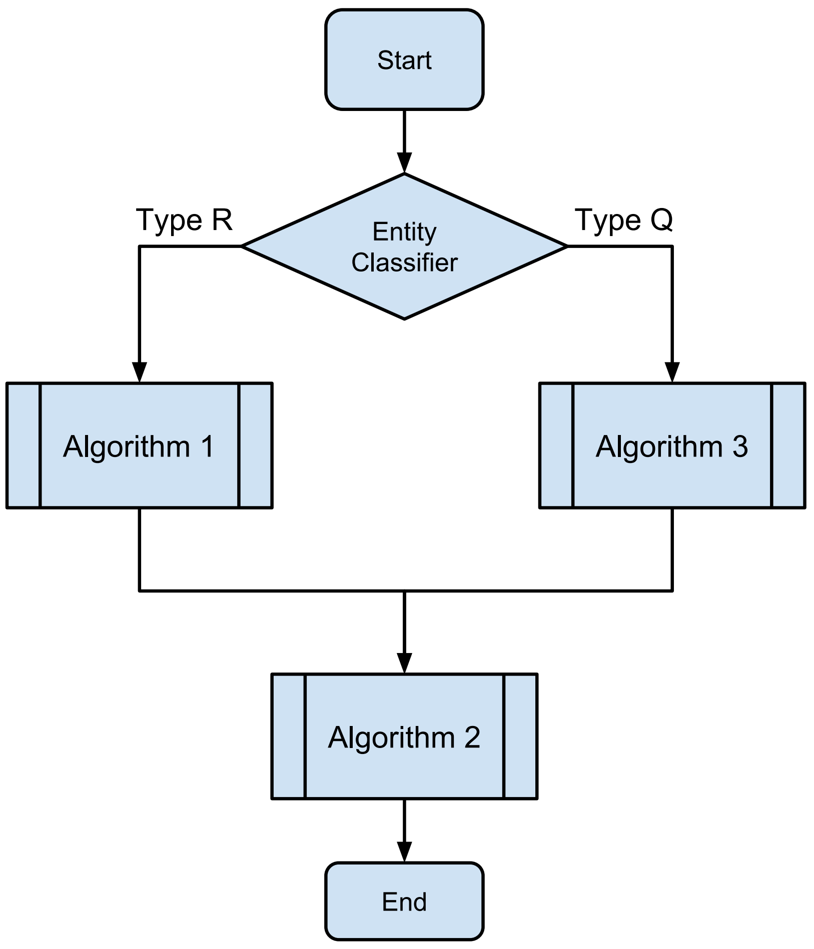


Figure 8.2: Workflow for executor

In the stand-alone feature recognizer, the DB systems can be replaced using hash tables directly. By using separate lists of local and non-local predicates, it can be avoid that everything going into nested loops. The approach proposed is based on what I have learned from the PostgreSQL testbed, so it should achieve the same performance. Although it is not implemented, I believe it provides a sound basis for interested system builders.

Bibliography

[AK91] S Anand and Kenneth Knott. An algorithm for converting the boundary representation of a CAD model to its octree representation. *Computers & Industrial Engineering*, 21(1):343–347, 1991.

[Arm00] Cecil Armstrong. *Djinn: a geometric interface for solid modelling: specification and report*. Information Geometers, 2000. ISBN: 978-1874728139.

[AU92] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. Computer Science Press, Inc., New York, NY, USA, 1992. ISBN: 978-0716782841.

[Bab16] László Babai. Graph isomorphism in quasipolynomial time iii: The Split-or-Johnson routine. `http://people.cs.uchicago.edu/~laci/quasipoly.html`, 2016. Retrieved 20 January 2016.

[BDS08] Emmanuel Brousseau, Stefan Dimov, and Rossitza Setchi. Knowledge acquisition techniques for feature recognition in CAD models. *Journal of Intelligent Manufacturing*, 19(1):21–32, 2008.

[BE77] Mike W. Blasgen and Kapali P. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):363–377, 1977.

[BNM08] Bojan Babic, Nenad Nesic, and Zoran Miljkovic. A review of automated feature recognition with rule-based pattern recognition. *Computers in Industry*, 59(4):321–337, 2008.

[BR09a] André Borrmann and Ernst Rank. Specification and implementation of directional operators in a 3D spatial query language for building information models. *Advanced Engineering Informatics*, 23(1):32–44, 2009.

[BR09b] André Borrmann and Ernst Rank. Topological analysis of 3D building models using a spatial query language. *Advanced Engineering Informatics*, 23(4):370–385, 2009.

[BS96] Geoffrey Butlin and Clive Stops. CAD data repair. In *Proceedings of the 5th International Meshing Roundtable*, pages 7–12. Citeseer, 1996.

[Bur10] Donald K Burleson. *Oracle tuning: the definitive reference*. Rampant TechPress, 2010. ISBN: 978-0982306130.

[CC91] J Corney and Doug ER Clark. Method for finding holes and pockets that connect multiple faces in 2 1/2D objects. *Computer-Aided Design*, 23(10):658–668, 1991.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Pioneers & Their Contributions to Software Engineering*, 13(6):377–387, 1970.

[Cor93] Jonathan Roy Corney. *Graph-based feature recognition*. PhD thesis, Heriot-Watt University, 1993.

[Cot09] J Cottrell. *Isogeometric analysis toward integration of CAD and FEA*. Wiley, Chichester, West Sussex, U.K. Hoboken, NJ, 2009. ISBN: 978-0470748732.

[CS95] Surajit Chaudhuri and Kyuseok Shim. An overview of cost-based optimization of queries with aggregates. *IEEE Data Engineering Bulletin*, 18(3):3–9, 1995.

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3):374–425, 2001.

[DG85] David J. DeWitt and Robert H. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, VLDB '85, pages 151–164. VLDB Endowment, 1985.

[DMFG94] T De Martino, B Falcidieno, and F Giannini. An adaptive feature recognition process for machining contexts. *Advances in Engineering Software*, 20(2):91–105, 1994.

[DuB05] Paul DuBois. *MySQL (3rd Edition) (Developer's Library)*. Sams, Indianapolis, IN, USA, 2005. ISBN: 0672326736.

[FA94] Malcolm C Fields and David C Anderson. Fast feature extraction for machining applications. *Computer-Aided Design*, 26(11):803–813, 1994.

[FOL$^+$03] MW Fu, Soh-Khim Ong, Wen Feng Lu, IBH Lee, and Andrew YC Nee. An approach to identify design and manufacturing features from a data exchanged part model. *Computer-Aided Design*, 35(11):979–993, 2003.

[GIS97] Paul Gibson, Hossam Ismail, and Malcolm Sabin. A feature recognition project. In *Product Modeling for Computer Integrated Design and Manufacture*, pages 179–190. Chapman & Hall, Ltd., 1997.

[GIS99] P Gibson, HS Ismail, and MA Sabin. Optimization approaches in feature recognition. *International Journal of Machine Tools and Manufacture*, 39(5):805–821, 1999.

[GISH97] P Gibson, HS Ismail, MA Sabin, and KKB Hon. Interactive programmable feature recogniser. *CIRP Annals on Manufacturing Technology*, 46(1):407–410, 1997.

[GP92] Rajit Gadh and Fritz B Prinz. Recognition of geometric forms using the differential depth filter. *Computer-Aided Design*, 24(11):583–598, 1992.

[Gra15] GrabCAD. GrabCAD Free CAD Models. `https://grabcad.com/library`, 2015. Retrieved 11 March 2015.

[GS98] Shuming Gao and Jami J Shah. Automatic recognition of interacting machining features based on minimal condition subgraph. *Computer-Aided Design*, 30(9):727–739, 1998.

[GS01] Ewald Geschwinde and Hans-Juergen Schonig. *PostgreSQL Developer's Handbook*. Sams Publishing, Indianapolis, IN, USA, 2001. ISBN: 978-0672322609.

[GZL$^+$10] Shuming Gao, Wei Zhao, Hongwei Lin, Fanqin Yang, and Xiang Chen. Feature suppression based CAD mesh model simplification. *Computer-Aided Design*, 42(12):1178–1188, 2010.

[HA84] Mark R Henderson and David C Anderson. Computer recognition and extraction of form features: a cad/cam link. *Computers in Industry*, 5(4):329–339, 1984.

[HCB05] Thomas JR Hughes, John A Cottrell, and Yuri Bazilevs. Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. *Computer Methods in Applied Mechanics and Engineering*, 194(39):4135–4195, 2005.

[Hip13] Richard Hipp. SQLite explain query. `http://www.sqlite.org/eqp.html`, 2013. Retrieved 9 July 2015.

[Hip15] D. Richard Hipp. The SQLite query planner. `https://www.sqlite.org/optoverview.html`, 2015. Retrieved 11 March 2015.

[HLGF04] Okba Hamri, Jean-Claude Leon, Franca Giannini, and Bianca Falcidieno. From CAD models to FE simulations through a feature-based approach. In *ASME 2004 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 377–386. American Society of Mechanical Engineers, 2004.

[HLL13] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2013.

[Hod14] Isaac Hodes. A deep dive into unexpectedly slow SQLite queries. `http://blog.isaachodes.io/p/deep_dive_into_slow_sqlite3_queries/`, 2014. Retrieved 9 July 2015.

[HPR00] JungHyun Han, Mike Pratt, and William C Regli. Manufacturing feature recognition from solid models: a status report. *IEEE Trans. Robotics and Automation*, 16(6):782–796, 2000.

[HR97] JungHyun Han and Aristides AG Requicha. Integration of feature based design and feature recognition. *Computer-Aided Design*, 29(5):393–403, 1997.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’93, pages 267–276, New York, NY, USA, 1993. ACM.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[IND15] PostgreSQL: CREATE INDEX. PostgreSQL index. `http://www.postgresql.org/docs/9.4/static/sql-createindex.html`, 2015. Retrieved 9 July 2015.

[Ioa96] Yannis E Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[ITI15] ITI Transcendata. CADfix. `http://www.transcendata.com/products/cadfix`, 2015. Retrieved 11 March 2015.

[JC88] Sanjay Joshi and Tien-Chien Chang. Graph-based heuristics for recognition of machined features from a 3D solid model. *Computer-Aided Design*, 20(2):58–66, 1988.

[Jon15] Jon Peddie Research. 2015 CAD report. `http://jonpeddie.com/publications/cad_report`, 2015. Retrieved 20 October 2015.

[Kim92] Yong Se Kim. Recognition of form features using convex decomposition. *Computer-Aided Design*, 24(9):461–476, 1992.

[KSL97] F.-L. Krause, C. Stiel, and J. Lüddemann. Processing of CAD-data – conversion, verification and repair. In *Proceedings of the Fourth ACM Symposium on Solid Modeling and Applications*, SMA '97, pages 248–254, New York, NY, USA, 1997. ACM.

[Kyp80] L K Kyprianou. *Shape classification in computer-aided design.* PhD thesis, University of Cambridge, 1980.

[LAPL05] KY Lee, Cecil G Armstrong, Mark A Price, and JH Lamont. A small feature suppression/unsuppression system for preparing B-rep models for analysis. In *Proc. 2005 ACM Symp. Solid and Physical Modeling*, pages 113–124. ACM, 2005.

[LCCT98] Gordon Little, Doug ER Clark, Jonathan R Corney, and JR Tuttle. Delta-volume decomposition for multi-sided components. *Computer-Aided Design*, 30(9):695–705, 1998.

[LG05] Helen L Lockett and Marin D Guenov. Graph-based feature recognition for injection moulding based on a mid-surface approach. *Computer-Aided Design*, 37(2):251–262, 2005.

[LS07] Shiqiao Li and Jami J Shah. Recognition of user-defined turning features for mill/turn parts. *Journal of Computing and Information Science in Engineering*, 7(3):225–235, 2007.

[Ltd15] Alpha Company Ltd. CPU heat sink. `http://www.micforg.co.jp/en`, 2015. Retrieved 18 March 2015.

[LZM14] Ming Li, Bo Zhang, and Ralph R Martin. Second-order defeaturing error estimation for multiple boundary features. *Int. J. Numerical Methods in Engineering*, 100(5):321–346, 2014.

[MB08] Kenton McHenry and Peter Bajcsy. An overview of 3D data content, file formats and viewers. *National Center for Supercomputing Applications*, 1205, 2008.

[McM00] Sara Anne McMains. *Geometric algorithms and data representation for solid freeform fabrication*. PhD thesis, University of California, Berkeley, 2000.

[Mel93] Jim Melton. *Understanding the new SQL : a complete guide*. Morgan Kaufmann Publishers, San Mateo, Calif, 1993. ISBN: 978-1558602458.

[MHS01] Sara McMains, Joseph M Hellerstein, and Carlo H Séquin. Out-of-core build of a topological data structure from polygon soup. In *Proceedings* *of the sixth ACM symposium on Solid modeling and applications*, pages 171–182. ACM, 2001.

[MK90] Michael Marefat and Rangasami L. Kashyap. Geometric reasoning for recognition of three-dimensional object features. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(10):949–965, 1990.

[ml15] PostgreSQL mailing list. PostgreSQL EXISTS optimization. `http://www.postgresql.org/message-id/25339.1174686582@sss.pgh.pa.us`, 2015. Retrieved 9 July 2015.

[MNS96] Martti Mäntylä, Dana Nau, and Jami Shah. Challenges in feature-based manufacturing research. *Communications of the ACM*, 39(2):77–85, 1996.

[Mol00] Hector Molina. *Database system implementation*. Prentice Hall, Upper Saddle River, N.J, 2000. ISBN: 978-0130402646.

[Mom01] Bruce Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley New York, 2001. ISBN: 978-0201703313.

[Mom12] Bruce Momjian. Explaining the PostgreSQL query optimizer. `http://pgday.ru/files/pgmaster14/bruce.momjian.optimizer.pdf`, 2012. Retrieved 11 March 2015.

[MSDS04] Madhu S Medichalam, Jami J Shah, and Roshan D Souza. N-rep: a neutral feature representation to support feature mapping and data exchange across applications. In *ASME 2004 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 599–609. American Society of Mechanical Engineers, 2004.

[MyS15a] MySQL. Creating spatial indexes. http://dev.mysql.com/doc/refman/5.0/en/creating-spatial-indexes.html, 2015. Retrieved 9 July 2015.

[MyS15b] MySQL. Mysql v5.6.4 release notes. http://dev.mysql.com/doc/relnotes/mysql/5.6/en/news-5-6-4.html, 2015. Retrieved 9 July 2015.

[NMS+15] Zhibin Niu, Ralph R Martin, Malcolm Sabin, Frank C Langbein, and Henry Bucklow. Applying database optimization technologies to feature recognition in CAD. *Computer-Aided Design and Applications*, 12(3):373–382, 2015.

[od15] PostgreSQL official document. Postgresql advantages. http://www.postgresql.org/about/, 2015. Retrieved 9 July 2015.

[ÖÖ01] Nursel Öztürk and Ferruh Öztürk. Neural network based non-standard feature recognition to integrate CAD and CAM. *Computers in Industry*, 45(2):123–135, 2001.

[PH92] Shashikanth Prabhakar and Mark R Henderson. Automatic form-feature recognition using neural-network-based techniques on boundary representations of solid models. *Computer-Aided Design*, 24(7):381–393, 1992.

[Pos15] PostgreSQL. The PostgreSQL index. http://www.postgresql.org/docs/9.1/static/sql-createindex.html, 2015. Retrieved 11 March 2015.

[PSB95] Mark Price, Clive Stops, and Geoffrey Butlin. A medial object toolkit for meshing and other applications. In *Proceedings of 4th International Meshing Roundtable*, pages 219–229. Citeseer, 1995.

[RGL90] Arnon Rosenthal and Cesar Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. *ACM SIGMOD Record*, 19(2):291–299, 1990.

[RGN97] William C Regli, Satyandra K Gupta, and Dana S Nau. Towards multiprocessor feature recognition. *Computer-Aided Design*, 29(1):37–51, 1997.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004. ISBN: 978-0262220695.

[RHJ$^+$01] Jeffrey Ricker, David Hurst, David Jakopac, Yerasi Chandrasekhara Reddy, and Kevin Kail. Data interchange format transformation method and data dictionary used therefor, July 2 2001. US Patent App. 09/896,125.

[SAKJ01] Jami J Shah, David Anderson, Yong Se Kim, and Sanjay Joshi. A discourse on geometric feature recognition from CAD models. *J. Computing and Information Science in Engineering*, 1(1):41–51, 2001.

[SAR94] JJ Shah, A Ali, and MT Rogers. Investigation of declarative feature modeling. *Computers in Engineering*, 1:1–1, 1994.

[SBRU95] Jami J Shah, G Balakrishnan, Mary T Rogers, and Susan D Urban. Comparative study of procedural and declarative feature based geometric modeling. In *Advanced CAD/CAM Systems*, pages 105–123. Springer, 1995.

[SC93] Hiroshi Sakurai and Chia-Wei Chin. Defining and recognizing cavity and protrusion by volumes. *Computers in Engineering*, pages 59–59, 1993.

[SC94] Hiroshi Sakurai and C Chin. Definition and recognition of volume features for process planning. *Manufacturing Research and Technology*, 20:65–65, 1994.

[SCC01] Raymond CW Sung, Jonathan R Corney, and Doug ER Clark. Automatic assembly feature recognition and disassembly sequence generation. *Journal of Computing and Information Science in Engineering*, 1(4):291–299, 2001.

[SD96] Hiroshi Sakurai and Parag Dave. Volume decomposition and feature recognition, Part II: curved objects. *Computer-Aided Design*, 28(6):519–537, 1996.

[Sha86] Leonard D Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems (TODS)*, 11(3):239–264, 1986.

[Sha91] Jami J Shah. Conceptual development of form features and feature modelers. *Research in Engineering Design*, 2(2):93–108, 1991.

[SHCK95] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and Richard Keefe. Logic programming for the real world. In *Proceedings of the ILPS*, volume 95, pages 83–94. Citeseer, 1995.

[SKG97] Ratnaker Sonthi, Girish Kunjur, and Rajit Gadh. Shape feature determination usiang the curvature region representation. In *Proc. 4th ACM Symposium on Solid Modeling and Applications*, pages 285–296. ACM, 1997.

[SM95] Jami J Shah and Martti Mäntylä. *Parametric and feature-based CAD/-CAM: concepts, techniques, and applications*. John Wiley & Sons, 1995. ISBN: 978-0471002147.

[SP09] VB Sunil and SS Pande. Automatic recognition of machining features using artificial neural networks. *The International Journal of Advanced Manufacturing Technology*, 41(9-10):932–947, 2009.

[SQL15a] SQLite. Sqlite analyze. `https://sqlite.org/lang_analyze.html`, 2015. Retrieved 1 September 2015.

[SQL15b] SQLite. Sqlite architecture. `https://www.sqlite.org/arch.html`, 2015. Retrieved 9 July 2015.

[SQL15c] SQLite. Sqlite subquery flattening optimization. `https://www.sqlite.org/eqp.html`, 2015. Retrieved 1 September 2015.

[SS91] Philip Spiby and Doug Schenck. EXPRESS language reference manual. *ISO TC184/SC4 Document N*, 14, 1991.

[SW95] Somashekar Subrahmanyam and Michael Wozny. An overview of automatic feature recognition techniques for computer-aided process planning. *Computers in Industry*, 26(1):1–21, 1995.

[SW97] Yong Seok Suh and Michael J Wozny. Interactive feature extraction for a form feature conversion system. In *Proceedings of the fourth ACM symposium on Solid modeling and applications*, pages 111–122. ACM, 1997.

[SWW$^+$12] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[Tan92] Zesheng Tang. Octree representation and its applications in CAD. *Journal of Computer Science and Technology*, 7(1):29–38, 1992.

[TDM$^+$10] Stefano Tornincasa, Francesco Di Monaco, et al. The future and the evolution of CAD. In *Proceedings of the 14th international research/expert conference: trends in the development of machinery and associated technology*, pages 11–18, 2010.

[The15a] The PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/about/`, 2015. Retrieved 11 March 2015.

[The15b] The PostgreSQL Global Development Group. PostgreSQL statistical information. `http://www.postgresql.org/docs/9.1/static/view-pg-stats.html`, 2015. Retrieved 11 March 2015.

[TK94] Sanjeev N Trika and Rangasami L Kashyap. Geometric reasoning for extraction of manufacturing features in iso-oriented polyhedrons. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(11):1087–1100, 1994.

[VR93] Jan H Vandenbrande and Aristides AG Requicha. Spatial reasoning for the automatic recognition of machinable features in solid models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(12):1269–1285, 1993.

[VR04] AK Verma and S Rajotia. Feature vector: a graph-based feature recognition methodology. *Int. J. Production Research*, 42(16):3219–3234, 2004.

[Wik15a] Wiki. Express data modeling language. `https://en.wikipedia.org/wiki/EXPRESS_(data_modeling_language)`, 2015. Retrieved 21 August 2015.

[wik15b] wikipedia. Comparison of relational database management systems. `http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems`, 2015. Retrieved 9 July 2015.

[wik15c] wikipedia. Manifold concept. `https://en.wikipedia.org/wiki/Manifold`, 2015. Retrieved 21 August 2015.

[wik15d] wikipedia. Programming language. `https://en.wikipedia.org/wiki/Programming_language`, 2015. Retrieved 21 August 2015.

[wik15e] wikipedia. Prolog. `https://en.wikipedia.org/wiki/Prolog`, 2015. Retrieved 21 August 2015.

[Win15] Markus Winand. Oracle execution plan operations. `http://use-the-index-luke.com/sql/explain-plan/oracle/operations`, 2015. Retrieved 16 August 2015.

[Woo82] Tony C Woo. Feature extraction by volume decomposition. In *Conference on CAD/CAM technology in mechanical engineering*, pages 76–94, 1982.

[ZM02] H Zhu and CH Menq. B-rep model simplification by automatic fillet/round suppressing for efficient automatic feature recognition. *Computer-Aided Design*, 34(2):109–123, 2002.