Applying Database Optimization Technologies to Feature Recognition in CAD

Zhibin Niu[1], Ralph R. Martin[2], Malcolm Sabin[3], Frank C. Langbein[4], and Henry Bucklow[5]

[1]Cardiff University, mind3str@gmail.com
[2]Cardiff University, ralph@cs.cf.ac.uk
[3]Numerical Geometry Ltd., malcolm.sabin@btinternet.com
[4]Cardiff University, frank@langbein.org
[5]TranscenData Europe Ltd. jhb@transcendata.com

ABSTRACT

In engineering analysis, CAD models are often simplified by removing features, enabling meshing to be quicker and more reliable; the resulting smaller meshes in turn lead to faster analysis. Finding features by hand is tedious, and there is a need to automate this process. A declarative approach to feature recognition allows engineers to define features relevant to a particular problem, without detailing how they are to be found. Here, we show that a declarative feature definition can be turned into an SQL query, and database engine coupled to a CAD modeler can be used to find instances of entities satisfying the predicates which make up features. A key benefit of doing so is that database optimization techniques built into a modern database can effectively execute the SQL query in an acceptable time to find features. We present experiments to show the benefits of various database optimization techniques. We determine how the time taken to find features scales with number of features and model size, using different optimizations. We also give results for real industrial models.

Keywords: computer-aided design, feature recognition, declarative language, SQL, database optimization

1. INTRODUCTION

Features are regions within an object's shape which are relevant to some point of view, such as its function or its manufacture. Computer-aided engineering (CAE) often performs analysis or simulation based on a CAD model, which requires meshing the model. Real industrial models have many small details, or features, and in many cases, their effect on analysis is minor. Suppressing such details allows meshing to be both quicker and more robust, and as a mesh with fewer, larger elements results, the time needed for analysis is also reduced. Feature recognition can be used to help find candidate features for removal [12]. Computer-aided process planning (CAPP) uses feature recognition to process low-level geometrical information such as edges and faces to derive manufacturing information in the form of high-level entities like holes and slots, as a precursor to generating a sequence of manufacturing instructions [8]. Finding features by hand is tedious for large models: automatically finding features is preferable. Feature recognition has been a topic of interest for many years and much research has been devoted to this topic [8].

Our approach to feature recognition is based on a high-level *declarative* feature definition language, which allows engineers to define new kinds of features. A declarative approach has the benefit of allowing the engineers to concentrate on *what* constitutes a feature, as they do not need to provide an algorithm saying *how* to find it. Different applications need different definitions of features: parts of a shape which are important for machining may be quite different to those which can be ignored for structural analysis, and features important for heat analysis may be quite different again. It is infeasible to hard-code all possible useful features, and this must be left to engineers.

In our method, features are defined in terms of entities (such as faces and edges, or subfeatures), and predicates (functions with Boolean results, e.g. whether two faces are adjacent) that the entities must satisfy. The definition is turned into an SQL query, and a database engine is coupled to a CAD modeler

© 2014 CAD Solutions, LLC, http://www.cadanda.com
Taylor & Francis
Taylor & Francis Group

to find instances of entities satisfying the predicates which make up features. The purpose of this paper is to show that such an approach is feasible, and that in particular, the database optimization techniques built in to a modern database can effectively execute the query in an acceptable time — a direct translation of the declarative form would be far too slow without optimization. Database optimization techniques automatically determine a suitable set of sub-queries and an appropriate order to execute them in. We build on the work of Gibson [6], who also used a declarative approach, but devised his own set of optimizations for use in a hand-coded feature finder, rather than leveraging database technology.

The rest of this paper is organized as follows: Section 2 briefly discusses related work on feature recognition and database optimization, as well reviewing Gibson's approach. Section 3 explains our declarative feature definition language, while Section 4 details our feature finder. Section 5 presents experimental results while Section 6 gives conclusions and discusses future work.

2. RELATED WORK

2.1. Feature Recognition

Martin gives the following definition: "A feature is a semantic group (modeling atom), characterized by a set of parameters, used to describe an object which cannot be broken down, used in reasoning relative to one or more activities linked to the design and use of products and production systems" [14]; also see [2]. Feature recognition is thus used to detect structures with certain topological and geometrical properties. It is distinct from partitioning the model, as features can overlap, and parts of the model may not be within any feature. Since the seminal work on geometric model analysis and classification by Kyprianou [11], much work has considered feature extraction. Babic's review [1] classifies approaches to feature recognition into approaches based on (i) syntactic pattern recognition, (ii) state transition diagrams, (iii) logic rules and expert systems, (iv) graph-based approaches, (v) convex hull and cell based volumetric decomposition, (vi) hint-based approaches, and (vii) hybrid approaches. In general, relations or connections between components of a feature are used to build up features from substructures. However, as noted, different applications need different definitions of features: parts of a shape which are important for machining differ from those which can be ignored for analysis, for example. It is infeasible to hard-code all possible useful features and engineers must be allowed to define new kinds of features for a task in hand.

Gibson thus proposed the use of declarative feature definitions [5–7], which have the benefit of stating what features are rather than needing an algorithm to find them. In his architecture, features are defined in a language with similarities to EXPRESS [15]. A feature-finding method directly derived from such a definition is generally infeasibly slow, however, so Gibson investigated six strategies for optimizing the feature finding process (as detailed later). These optimizations demonstrated significant benefits on 2D models. Our work builds on Gibson's work, again using a declarative feature definition language. However, we convert the feature definition into an SQL database query which is then input into a feature finder which combines a CAD modeler with a relational database. Note that SQL is also a declarative language. In this way, our aim is to leverage the extensive research into query optimization which is embodied in modern databases. Primitive geometric and topological queries are passed from the database engine to the CAD modeler.

2.2. Relational Database Query Optimization

In a relational database, the same query can typically be translated in many different ways into a sequence of elementary operations; although the output is identical, the time taken can differ greatly [10]. Our system automatically translates the user's feature definition into an SQL query. Within the database this SQL string is analyzed into tokens (e.g., `SELECT FROM`) and assigned meaning by a parser, turning the feature definition into a relational calculus expression. The query optimizer further transforms the expression tree into several different forms (i.e. possible query plans) with equivalent results, and determines which is likely to be most efficient by estimating the cost. Query optimizers are a core part of database management systems, and have been widely studied. Among them, System-R was a pioneer [4], being one of the earliest databases to support SQL. Its use of dynamic programming to select the best query plan has been adopted by most commercial databases [3].

Query optimization comprises two stage: rewriting and planning [10]. The former rewrites the declarative query in the expectation that the new form may be more efficient. An example of this kind is *sargable* rewriting (i.e. transformation to take advantage of an index). Planning transforms the query at a procedural level, using relational algebra transformations [4] such as generalizing join sequences, outer join optimization, group-by and join shuffling optimizations, view merging, merging nested subqueries, and using semi-join-like techniques to optimize multi-block queries. The most important job of the query planner is to determine which form of the query generated by such transformations is most efficient.

We have chosen to use SQLite as the database engine in our system, as source code is readily available. This allows it to be interfaced to the CAD modeler, in this case CADfix [13]. It has a compact but effective query optimizer [9]: it provides sargable rewriting including `BETWEEN` and `OR` optimizations, and provides algebraic space and method-structure space transformations such as reordering

© 2014 CAD Solutions, LLC, http://www.cadanda.com

joins, subquery flattening, automatic indexing and group-by optimizations. SQLite 3.8 uses a *nearest neighbor* heuristic leading to an efficient polynomial-time algorithm to select the best plan.

2.3. Comparison with Gibson's Optimization Approach

In our approach, feature definitions are essentially translated into nested for-all loops, iterating over each named entity. The loops check the predicates and output only combinations of entities meeting the predicates, as directed by export statements. In principle, all possible combinations of entities and predicates should be tested. Directly doing so would take a time exponential in the number of entities, so is infeasible in all but the most trivial of cases.

Gibson also adopted a declarative approach to feature finding, and suggested six ways to reduce computational complexity [6]. These are

Strength reduction Loop re-sequencing Entity classification Assignment Indexing Featuretting.

Instead, we rely on optimization methods built-in to databases to enable feature finding to run at a suitable speed. In fact, in part, Gibson has rediscovered (or reinvented) a subset of the kinds of optimizations used to answer database queries, as we now explain.

Gibson's strength reduction moves tests with fixed results outside loops to avoid repeatedly computing the same result. Loop re-sequencing reorders the order in which nested loops are executed. Similar steps are used in database query processing, where they are referred to as algebraic transformations which re-order joins and tables.

Indexing is a further significant optimization considered in database optimization and by Gibson. However, its use in databases is more advanced: for example, SQLite supports automatic index creation, and uses sophisticated forms of indexing such as R−/R+ tree indexes. Other databases such as PostgreSQL go further and support B-tree, hash, GiST, and GIN indexes. We intend to investigate indexing more fully in our future work.

Gibson also has some unique approaches which are not used in database engines. The first one is entity classification which splits the for-all loop into two parts: a simple search with only one predicate and the remaining parts in which the simple predicate is replaced by a result list. This allows preprocessing of some basic relations e.g., size relations, and uses the results list to replace the full relation. Featuretting is similar to entity classification, where a subfeature query and a 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 perform searches locally. Furthermore, the root feature has to have at least two WITH clauses if featurettes are to be used. On the other hand, databases generally process a complete query; even if there are subqueries, the database does not preprocess them, but flattens the input into a long query. This allows indexes to be used locate results. Finally, in Gibson's assignment optimization, when WHERE clauses contain equality conditions, nested loops to find two things which are then set equal can be replaced by a single loop. This treats equality predicates as assignment statements to reduce the search space. This is also unused in database query optimization.

Database optimization relies heavily on cost estimation to choose the best transformation of the original SQL request to deduce a query plan. However, Gibson's methods correspond more to a rule-based system which rewrites the feature definition at the declarative level. From the point of view of transformations, Gibson's strength reduction and loop re-sequencing are forms of join sequencing used in database optimization. Some database optimizations have no equivalents in Gibson's work, e.g. outer join optimization [4] which reorders joins and outer joins for speed. Merging nested subqueries is an important rewriting method to deal with sub-conditions, while in Gibson's work, similar expressions (subfeatures) are handled by entity classification and featuretting.

In summary, while Gibson's approach and use of database optimization clearly have some ideas in common, they also have differences, and each considers some ideas the other lacks. This is because they were originally intended for solving different problems; Gibson's ideas are specific to feature finding in a CAD system, while database query optimization must deal with a very wide range of queries.

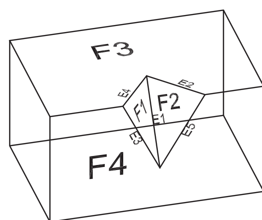
3. DECLARATIVE FEATURE DEFINITION

We next explain how features are defined in our system and the form of our declarative language. As noted, a declarative approach has the benefit that the engineer can clearly separate 'what a particular feature is' from the more difficult issue of 'how to find such a feature'. The declaration is automatically converted into a naive algorithm for funding the feature, and then the query optimizer tackles the harder job of turning that into an efficient algorithm with reduced complexity. We adopt the general form of feature definition shown below.

```
DEFINE FEATURE NAME AS ENTITY1, ENTITY2...: TYPE; ...
SATISFYING PREDICATE(ENTITY1, ENTITY2, ...); ...
EXPORT ENTITY1, ENTITY2....END
```

First, a name is given for this kind of feature. Next, various entities are defined which must be present in the feature. Each entity is either of a CAD modeler primitive data type, such as a vertex, edge, face, or surface, or is a sub-feature e.g. a CYLINDRICAL-HOLE. This type restricts the search domain. For example E1,E2:EDGE means E1 and E2 are edges, and

© 2014 CAD Solutions, LLC, http://www.cadanda.com




```
DEFINE NOTCH AS
      F1,F2,F3,F4:face; E1,E2,E3,E4,E5:edge;
SATISFYING
      BOUNDS(E1,F1);  BOUNDS(E1,F2);   BOUNDS(E2,F2);   BOUNDS(E2,F3);
      BOUNDS(E3,F1);  BOUNDS(E3,F4);   BOUNDS(E4,F1);   BOUNDS(E4,F4);
      BOUNDS(E5,F2);  BOUNDS(E5,F4);   LOWER_ID(F1,F2); LOWER_ID(F3,F4);
      DIFFERENT_ID(E2,E1);     DIFFERENT_ID(E3,E1);
      DIFFERENT_ID(E4,E1);     DIFFERENT_ID(E5,E1);
      DIHEDRAL(E1,CONCAVE);    DIHEDRAL(E2,CONVEX);
      DIHEDRAL(E3,CONVEX);     DIHEDRAL(E4,CONVEX);
END
```

Fig. 1: Notch feature and definition.

```
DEFINE TRIFACE AS F:FACE SATISFYING VALENCY(F,3) EXPORT F END
DEFINE TRIBOUND AS F:TRIFACE, E:EDGE SATISFYING BOUND(E,FACE) EXPORT F,E END
DEFINE TRIFACEPAIR AS F1,F2,F1E1,F1E2,F2E1,F2E2:TRIBOUND SATISFYING LOWER_ID(F1.F,F2.F);
LOWER_ID(F1.E,F2.E); SAME_ID(F1E1.F,F1.F); DIFFERENT_ID(F1E1.E,F1.E); LOWER_ID(F1E1.E,F1E2.E);
LOWER_ID(F1E2.F,F1.F); DIFFERENT_ID(F1E1.E,F1.E); SAME_ID(F2E1.F,F2.F); DIFFERENT_ID(F2E1.E,F2.E);
LOWER_ID(F2E1.E,F2E2.E); DIFFERENT_ID(F2E2.E,F2.E)  EXPORT F1.F AS FACE1, F2.F AS FACE2,
F1.E AS COMEDGE, F1E1.E AS F1E1, F1E2.E AS F1E2, F2E1.E AS F2E1, F2E2.E AS F2E2; END
DEFINE NOTCH AS P:TRIFACEPAIR  SATISFYING DIHEDRAL(P.COMEDGE,CONCAVE); DIHEDRAL(P.F1E1,CONVEX);
DIHEDRAL(P.F1E2,CONVEX); DIHEDRAL(P.F2E1,CONVEX); DIHEDRAL(P.F2E2,CONVEX); END
```

Fig. 2: Alternative notch definition.

that all edges of the model should be considered as candidates for E1 and E2.

The predicates after SATISFYING refer to one or more entities, and indicate various relationships that the entities should satisfy. Single entity predicates concern geometric properties e.g. DIHEDRAL(EDGE1, CONVEX), and continuity e.g. DIHEDRAL(EDGE1, TANGENTIAL), while binary predicates concern such issues as equality of entities e.g. DIFFERENT_ID (EDGE1,EDGE2), and bounds e.g. BOUNDS(EDGE1, FACE1) (meaning EDGE1 is part of the boundary of FACE1. Currently, the system can handle the entities FACE, EDGE, POINTS, SURFACE and has predicates including: VALENCY,BOUND,DIHEDRAL, LOWER_ID, DIFFERENT_ID, GEOM, HIGHER_ID, LIES_IN.

The EXPORT clauses indicates which entities are externally visible to other definitions. This allows a feature declaration to be built up using (hierarchical) subfeatures; exporting only relevant entities of subfeatures limits the amount of data to be processed.

To illustrate these ideas, Fig. 1 gives the definition for a notch feature. All edges and faces of the notch feature must be present, as must be the adjacent faces. The bounds predicates filter out edges of the model which do not belong to notch faces. The LOWER_ID relationships 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). The DIFFERENT_ID predicates prevent unwanted solutions where the same entity is found for things which should obviously be distinct. The convexity predicates are essential characters of a notch. Only if all 5 edges and 4 faces agree with the various predicates can the feature finder declare the presence of a notch feature.

Note that the declarative approach does not lead to unique definitions—the same feature may be defined in more than one way, and without optimization, different definitions would 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 we show later), and so the engineer needs to help. (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 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.

For example, a notch feature includes two adjacent triangular faces. We can define a 'triangle-face-pair' as a subfeature. Triangular faces are further subfeatures of a triangle-face-pair. Using subfeatures has similar effects to Gibson's (automatic) featuretting [6], an optimization technique used to ensure that each search only applies to a local domain, By doing so, later searches only need consider a smaller set of entities, resulting in lower computational complexity.

Fig. 2 gives another way to describe a notch feature as a series of subfeatures. Notches are defined in terms of triangle pairs. Triangle pairs are based upon triangle bounds. Triangle bounds depend upon triangle faces. These are found and remembered in a table, and information is propagated back up the hierarchy. Lower level subfeatures are more general and have simpler predicates.

4. FEATURE FINDER

We now describe how our system uses declarative feature definitions to find features. We start with an overview, and then consider further details.

© 2014 CAD Solutions, LLC, http://www.cadanda.com

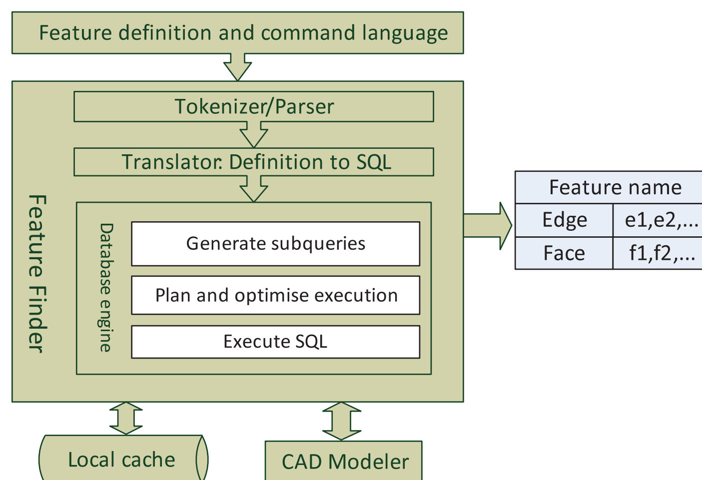


Fig. 3: Feature recognition framework.

4.1. Overview

The interface to the feature finder works is a command interpreter which interactively reads and executes imperative user commands. These are used to define a feature, load a CAD model, find instances of a feature, etc. After instances of a feature are found, details such as its edges and faces may be output, either as text, or as highlighting on a drawing of the original 3D model.

The structure of our feature finder is shown in Fig. 3. It is built around SQLite, an open-source database, and calls CADfix, a CAD modeler. The front end reads and interprets the command languages. A local cache of database tables is used to store certain frequently used data such as entities which may be components of the final feature.

The core of the feature finder includes a tokenizer and parser system to analyze the user's input. A translator turns feature definitions into SQL queries; these are passed to the database engine for execution when a request is made to find features. When the database engine executes the SQL query (after optimization), requests for basic geometric and topological information are passed to the CAD modeler (rather than being looked up in a table on disk, like a database engine would normally do); such information may be cached, to save repeatedly querying the CAD modeler. Other commands may be passed directly to the CAD modeler, such as requests to load CAD models, or be executed directly, such as requests to save feature definitions to disk. A list of commands is shown in Fig. 4.

In detail, when the user issues a `FIND` command, assuming a corresponding feature definition has been entered, the SQL query is sent to the database engine, where it is broken down into a set of simple queries concerning entities and predicates, and it is decided whether these should be answered by the CAD modeler, or the local database cache. The query is processed by a rewriter, and transformed by the algebraic space module and method structure module separately. The planner then chooses the plan expected to be cheapest to execute, to retrieve all entities which satisfy the predicates and hence correspond to features. The command `PRINT` can output textual details of the features found, or `SHOW` can be used 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.

Fig. 4: Command language.

There are various ways to rewrite SQL queries and to transform them in algebraic space, i.e. diverse ways to express the same kind of feature, as we consider further in the next section. Unsurprisingly, alternative plans differ in efficiency, as we show in our experiments.

4.2. Converting a Declaration Into SQL

A translator is used to turn a feature definition into SQL; both are expressed declaratively, so this is relatively straightforward. When the translator reads `DEFINE...AS`, it recognizes... as the feature type. The clause `AS...SATISFYING` is translated into an SQL `SELECT...FROM` clause. Within it, the types of entities such as `FACE` and `EDGE` are basic information provided by the CAD modeler. The predicates in the clause `SATISFYING...EXPORT` are turned into a `WHERE` clause, with predicates connected by `AND`. Finally, `EXPORT...END`, determines the output, which is specified using `SELECT`. As an example, one way

© 2014 CAD Solutions, LLC, http://www.cadanda.com

```
SELECT F1.FACE, ..., F4.FACE, E1.EDGE, ..., E5.EDGE FROM
    FACES AS F1, ..., FACES AS F4, EDGES AS E1, ..., EDGES AS E5
WHERE
    F1.FACE<F2.FACE AND F3.FACE<F4.FACE AND
    (E1.EDGE<>E2.EDGE) AND (E1.EDGE<>E3.EDGE) AND
    EXISTS (SELECT BOUNDS.EDGE FROM BOUNDS WHERE BOUNDS.FACE=F1.FACE AND BOUNDS.EDGE=E1.EDGE) AND
    EXISTS (SELECT BOUNDS.EDGE FROM BOUNDS WHERE BOUNDS.FACE=F2.FACE AND BOUNDS.EDGE=E1.EDGE) AND
    ... AND
    EXISTS (SELECT BOUNDS.EDGE FROM BOUNDS WHERE BOUNDS.FACE=F2.FACE AND BOUNDS.EDGE=E5.EDGE) AND
    EXISTS (SELECT BOUNDS.EDGE FROM BOUNDS WHERE BOUNDS.FACE=F4.FACE AND BOUNDS.EDGE=E5.EDGE);
```

Fig. 5: Notch definition as SQL.

to rewrite the original notch definition is shown in abbreviated form in Fig. 5:

Lists of all faces, edges, and vertices and their bounding relationships are repeatedly used in searching, so they are cached as tables of faces, edges, etc. Alias names are given to multiple entities from the same table, for example using `F1,F2,...` for `FACE` entities. This allows such must-have entities to be expressed via a self-join structure in SQL, generating the search space for the query. The predicates are expressed using `WHERE` clauses, and work as filters to retain only those records that meet the specified criteria. Predicates are translated into SQL clauses one by one following the feature definition. For example, the inequality predicate `DIFFERENT_ID` is translated into `<>`. Predicates like `BOUNDS(E1,F1)` are converted to `EXISTS(SELECT BOUNDS.EDGE...` clauses as shown above. The Boolean result indicates whether the desired BOUNDS relation exists or not.

4.3. Tools

SQLite has been chosen as the database engine in our system, for the pragmatic reasons that it free, has open source, and has clearly structured code which facilitates linking it to the CAD modeler to build a feature recogniser. 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.

CADfix is 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 supports import of CAD models in multiple formats. It already provides some defeaturing tools, although we do not make use of these. We use CADfix (via its API) to read and repair CAD models (to ensure consistent, connected topology), and to interrogate their topology and geometry. It is also used to draw the features found, and could be used for further processing such as defeaturing.

5. EXPERIMENTS

We 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, we consider three questions: does database optimization help, and if so how much? How powerful is SQLite database optimization? Is this relevant to real models?

5.1. Benefits of Database Optimization

In declarative feature recognition, as already noted, directly finding a feature from a definition uses nested loops, with time complexity $O(n^k)$ in a model with n entities, for features 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.

In the first experiment we generated a family of artificial models with an increasing number of blocks, each containing a notch feature of the kind defined earlier (see Fig. 6). A similar experiment was also performed using through-hole features. The notch definition has 9 entities and 17 predicates, while the through-hole definition has 11 entities and 24 predicates.

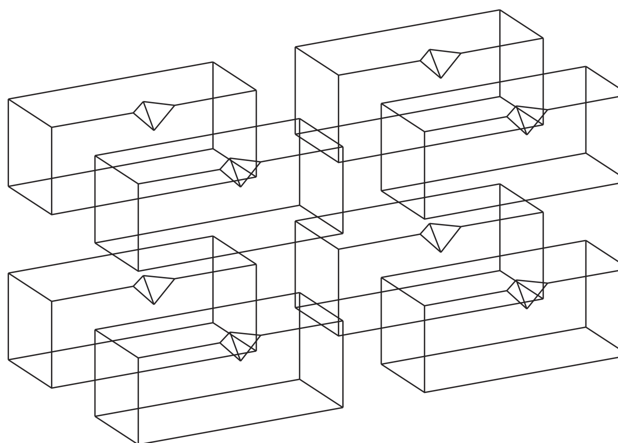


Fig. 6: Model with 8 notched blocks.

SQLite was modified to allow its built-in optimization approaches (re-ordering joins, sub-query flattening, and automatic indexing) to be turned off and on, to understand their effects on performance. Fig. 7 considers the models in a family, and gives a log-log

© 2014 CAD Solutions, LLC, http://www.cadanda.com

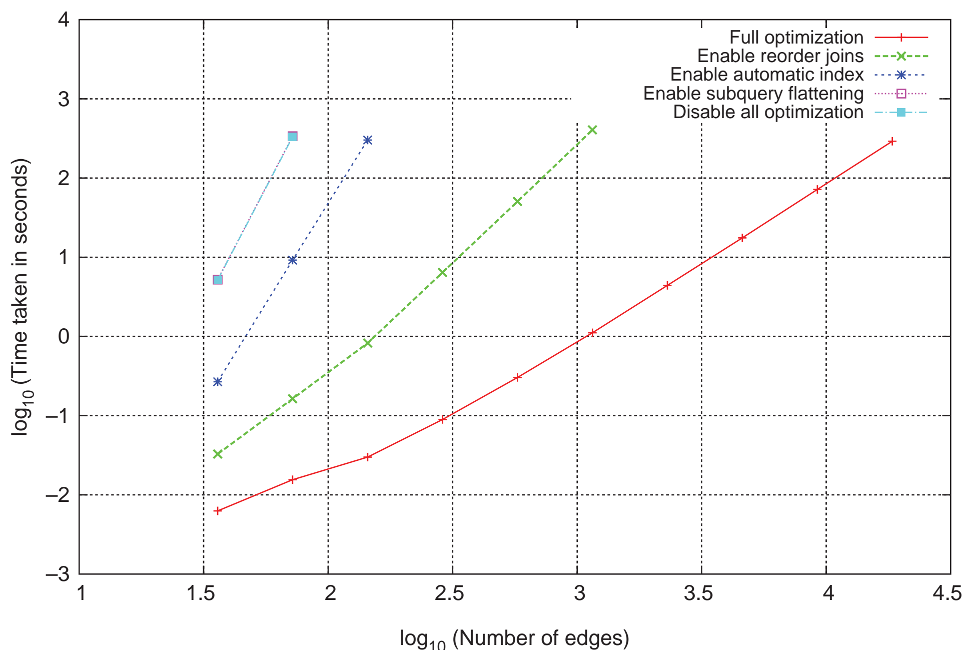


Fig. 7: Query optimization performance compare.

plot of time taken in seconds to find the features of the given type in each model, versus the total number of edges in that model. Turning all optimizations off clearly gives the worst performance, while turning all optimizations on is best, as hoped. 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 is used is reordering joins, dropping to a model of 70 edges if no optimization is used. Subquery flattening has little effect in this experiment and the curve when this is the only optimzation used is almost coincident with the un-optimized result.

While clearly the benefits of query optimization depend on the model size, already in this example they have enabled us to process models which are 250 times larger. A better way to quantify this is to consider the slope of the log-log plot which gives the exponent p assuming that the time taken by each version of feature finding is dominated by $t = \alpha n^p$.

Slopes using different optimization approaches for the notch feature and through-hole feature are shown in Tab. 1. For the notch feature, the fully optimized program has (approximately) a time complexity $O(n^2)$, while the un-optimized one has a much higher complexity, $O(n^6)$. 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. We can loosely say, for notch features, that we get 1 order of complexity of benefit from indexing and 3 orders from reordering joins. For through hole features, the query optimization provided less benefit, due to the somewhat different definition used, reducing complexity from about $O(n^{3.7})$ to $O(n^{2.3})$.

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

Tab. 1: Slopes for various optimizations.

The estimated slopes allow us to predict how time taken varies as model size increases. A model with a single notch takes about 5 seconds to process, unoptimized, while even the model shown with 8 notches would take 5×8^6 seconds $\approx$ 2 weeks, illustrating our assertion that a declarative approach without optimization is infeasible. On the other hand, with all optimizations turned on, a much larger model (about 18000 edges) can be analyzed in a feasible time (about 5 minutes). Reordering joins is the most powerful optimization, then automatic indexing is also useful, but sub-query flattening has little effect.

5.2. Optimality of Database Optimization

A separate question we can ask concerns how powerful database optimzation is at turning a sub-optimal query into an optimal query plan. As noted, there are various ways the same feature may be defined, which

© 2014 CAD Solutions, LLC, http://www.cadanda.com

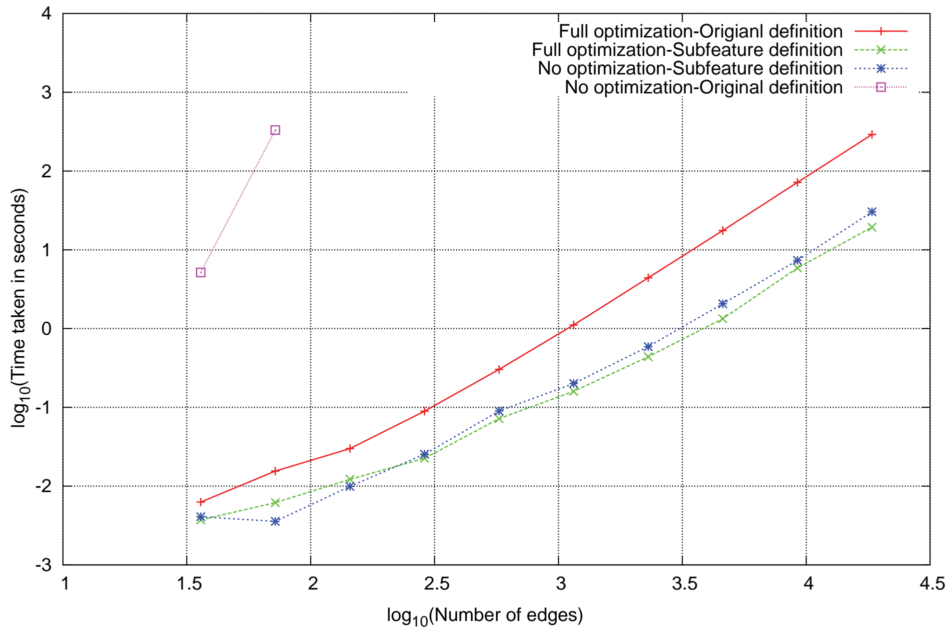


Fig. 8: Using alternative notch feature definitions.

give equivalent output, but do not take the same time to run; indeed, that is the whole basis for optimzation. Section 3 introduced just two ways to define a notch feature, using an all-in-one definition, and a subfeature-based definition. Now, if the database optimizer were powerful enough, then in principle it should be able to optimize both definitions to the same optimal form, and hence both would take the same time after optimiztion. Thus, we conducted a second experiment to determine to what extent the input feature definition affects the final optimized performance. We again used notch features. Query optimization was turned on or off for both definitions discussed earlier. Fig. 8 and Tab. 2 give the results of this test.

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

Tab. 2: Slopes for various definitions.

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, and hence the slope of the log time complexity graph if no optimization is done. The original notch definition is made up of 9 of entities, so the all-in-one definition if executed naively would have time complexity $O(n^9)$; in practice, we somewhat lower. The subfeature approach's naive time complexity is ultimately $O(n^6)$, and again we see better in practice, even with optimization turned off.

Nevertheless, we can see 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, we can draw two further conclusions. 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.

5.3. Real Industrial Models

Previous experiments in this paper considered artificial models, and real industrial models may be very complex and cause feature finders to behave differently. For example, in our 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

© 2014 CAD Solutions, LLC, http://www.cadanda.com

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, we used it to find SLOT features on a CPU heatsink, a carbine, and a switch, as shown in Figs 9–11. Results are summarized in Tab. 3.

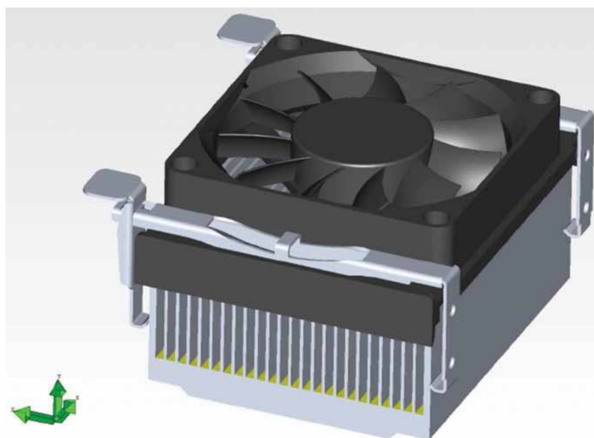


Fig. 9: CPU heatsink.

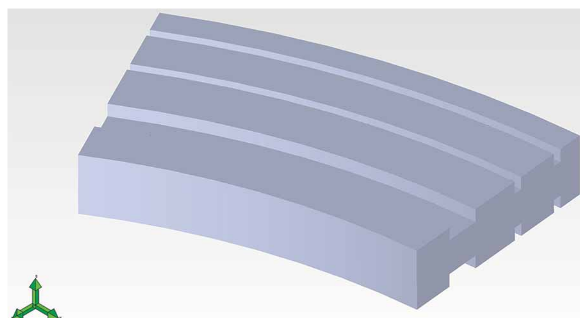


Fig. 10: Carbine.

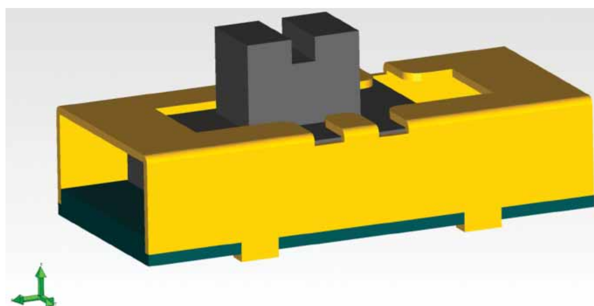


Fig. 11: Switch.

For the simplest model, the carbine, optimized feature finding took only 0.05s; without optimization the time taken was over 14 hours. The more complex switch and CPU heatsink models required 0.2s and 7s 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.

Model	CPU heatsink	Carbine	Switch
Number of edges	2388	84	330
Number of slots	24	6	9
Optimized query	6.94s	0.05s	0.22s
Unoptimized query	–	52092s	–

Tab. 3: Time taken to find slots in real models.

6. CONCLUSIONS AND FUTURE WORK

This work has considered a declarative approach to feature recognition, coupled with database optimization to enable such definitions to be processed in an acceptable time. An advantage of this approach over the similar earlier approach by Gibson is that we get ‘for free’ all the insight that has gone into database optimization. Results show that, as hoped, database optimization provides significant improvements to the time complexity of feature finding, leading to results in an acceptable time. However, optimization does not always provide an optimal result: different ways of defining the same feature have different performance even after optimization. We note that SQLite is a light-weight database, and does not support some advanced features such as recursive SQL and various kinds of advanced indexing; it also has more powerful optimizations. Our next steps are to transfer the current workbench to PostgreSQL and to investigate the benefits of its stronger optimization, advanced indexing, and recursive SQL.

ACKNOWLEDGEMENTS

This work was supported by the Framework Programme 7 Initial Training Network Funding under Grant No. 289361 “Integrating Numerical Simulation and Geometric Design Technology”.

REFERENCES

[1] Babic, B.; Nesic, N.; and Miljkovic, Z.: A review of automated feature recognition with rule-based pattern recognition, Computers in Industry, 59(4), (2008), 321–337.

[2] Babic, B. R.; Nesic, N.; and Miljkovic, Z.: Automatic feature recognition using artificial neural networks to integrate design and manufacturing: Review of automatic feature recognition systems., AI EDAM, 25(3), (2011), 289–304.

[3] Chamberlin, D. D.; Astrahan, M. M.; Blasgen, M. W.; Gray, J. N.; King, W. F.; Lindsay, B. G.;

© 2014 CAD Solutions, LLC, http://www.cadanda.com

Lorie, R.; Mehl, J. W.; Price, T. G.; Putzolu, F.; et al.: A history and evaluation of System R, Communications of the ACM, 24(10), (1981), 632–646.

[4] Chaudhuri, S.: An overview of query optimization in relational systems, in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, ACM, 1998, pages 34–43.

[5] Gibson, P.; Ismail, H.; and Sabin, M.: A feature recognition project, in *Proceedings of the fifth IFIP TC5/WG5. 2 international workshop on geometric modeling in computer aided design on Product modeling for computer integrated design and manufacture*, Chapman & Hall, Ltd., 1997, pages 179–190.

[6] Gibson, P.; Ismail, H.; and Sabin, M.: Optimisation approaches in feature recognition, International Journal of Machine Tools and Manufacture, 39(5), (1999), 805–821.

[7] Gibson, P.; Ismail, H.; Sabin, M.; and Hon, K.: Interactive programmable feature recognisor, CIRP Annals-Manufacturing Technology, 46(1), (1997), 407–410.

[8] Han, J.; Pratt, M.; and Regli, W. C.: Manufacturing feature recognition from solid models: a status report, Robotics and Automation, IEEE Transactions on, 16(6), (2000), 782–796.

[9] Hipp, R.: Sqlite optimizer, https://www.sqlite.org/optoverview.html, 2013.

[10] Ioannidis, Y. E.: Query optimization, ACM Computing Surveys (CSUR), 28(1), (1996), 121–123.

[11] Kyprianou, L. K.: Shape classification in computer-aided design., Ph.D. thesis, University of Cambridge, 1980.

[12] Lee, K.; Armstrong, C. G.; Price, M. A.; and Lamont, J.: A small feature suppression/unsuppression system for preparing b-rep models for analysis, in *Proceedings of the 2005 ACM symposium on Solid and physical modeling*, ACM, 2005, pages 113–124.

[13] Ltd, T. E.: CADfix 9.0., http://www.transcendata.com/products/cadfix, 2013.

[14] Martin, P.: Some aspects of integrated product and manufacturing process, in A. Bramley; D. Brissaud; D. Coutellier; and C. McMahon, editors, *Advances in Integrated Design and Manufacturing in Mechanical Engineering*, pages 215–226, Springer Netherlands, 2005.

[15] Spiby, P. and Schenck, D.: Express language reference manual, ISO TC184/SC4 Document N, 14.

© 2014 CAD Solutions, LLC, http://www.cadanda.com