An ASCII Database for Fast Queries of Relatively Stable Data

Appears in *Computing Systems*, Volume 4, Number 2

Eric H. Herrin II Raphael Finkel

Spring, 1991

Abstract

This paper describes `qddb`, a database program suite that addresses the common situation in which data are infrequently changed. Searches in the stable parts of databases are very efficient because they use hash-based searching through a completely inverted index, whereas searches in updated sections are less so. All files used by `qddb` are in Ascii format, so they can be easily modified by a knowledgeable user with a simple text editor. The Ascii format does not detract significantly from the efficiency of our database package, and it allows databases to be distributed and completely portable between architectures. The `qddb` suite currently runs under both BSD and System V Unix^{TM} on a variety of machines. It is capable of querying large databases on small machines with limited memory.

1 Introduction

`Qddb` is a database suite that attempts to provide a limited amount of function with high efficiency. It incorporates the following unusual ideas.

- A combination of hashing and indexing is used for quick retrieval. A complete inverted index is searched by hashing. Updates create data outside the scope of the hash table that must be searched in a less efficient manner.
- Although its exterior appears similar to other packages utilizing the relational model, `qddb` allows attribute values to be missing or multiple and of any length.
- Attribute values are stored as arbitrary-length Ascii strings. Ordinary text editors are used as data-entry and update tools.
- Attributes can be structured to contain other attributes.

`Qddb` remedies many inefficiencies of general database packages when the data are relatively stable, that is, when the data are changed infrequently. We concentrate on minimizing the number of system calls (*open* and *read* calls) per query, since they are significantly more time-consuming than small in-memory searches and computation. Our file structure is designed to minimize system calls in the case of a stable relation, but it still performs acceptably for a relation with instable parts.

A **stable** relation is one that has not been updated since the last stabilization. We provide a **stabilization** routine that parses the entire relation into keys and builds a hash table for quick retrieval of tuples that contain any given key. This process may require a significant amount of time in comparison to other database operations, but the time is not excessive. An **instable** relation is one that has been updated after the last stabilization; the **instable parts** of a relation are the tuples that have been added or updated since stabilization.

A **key** is defined to be any sequence of characters delimited by white space or punctuation in any attribute value of a tuple. This definition is not crucial to our approach, but serves as a useful convention. We could have been more restrictive (to uncommon English words) or more inclusive (to all strings). Retrieval time is not affected by how restrictive we are, but the size of one of the auxiliary files and the time to stabilize is dependent on how many distinct keys appear in a relation.

This paper begins by discussing the syntax and semantics of the files that comprise a relation. We analyze the operations on these files to demonstrate the performance benefits of our approach. The in-memory structures used by `qddb` programs are presented and analyzed to show how we use them to implement `QSQL`, a version of the query language `SQL` appropriate to `qddb` semantics. We then turn to enhancements of the basic model. We show how extra structure files can make some searches faster. Finally, we discuss drawbacks to our approach and how an implementor might overcome them.

2 `Qddb` semantics and syntax

The structure of each relation is defined by its **schema**, which specifies the attributes associated with tuples in that relation. A `qddb` **relation** is a set of tuples. Each **tuple** is a set of **attribute values**. Attributes are either simple or expandable. **Simple attributes** contain at most one value per tuple. An **expandable attribute** may have multiple values of the same tuple. We display these ideas in Figure 1.

In this example, attributes marked with "*" are expandable, that is, they may be given multiple values. A student may have multiple addresses; each address may have multiple phone numbers. Each student is expected to have taken multiple courses. Each occurrence of an expandable attribute is called an **instance** of that attribute. This example also shows that attributes may have sub-attributes. The *Address* attribute contains sub-attributes for such items as street address, city, and zip code, and *Street* itself has sub-attributes for number, road, and apartment number.

Tuples need not contain values for all defined attributes. For example, we might leave off the *Country* attribute for addresses in the same country as the school, or the *MiddleName* attribute for students with no middle name. No storage is occupied by attributes that have no value.

The most fundamental query supported by `qddb` returns all tuples containing a given key (independent of the attribute in which it is found). More complex queries are also supported, such as Boolean combinations, attribute-specific queries, and queries based on numeric or other properties of attributes.

2.1 Schema syntax

The `Schema` file specifies the names and order of a relation's attributes. It can be constructed with any ordinary text editor. Once the relation has been populated, the schema should be changed only by (1) adding new attributes at the end, (2) adding subattributes at the end of an attribute, or (3) using a program in the `qddb` suite that converts relations from one schema to another by translating the entire relation into presentation format by the first schema and then translating it back to storage format by the second one. We discuss these forms shortly.

The BNF grammar in Figure 2 defines the format of a schema file. Items in single quotes are terminals of the grammar. White space in the file is used as a separator and is otherwise ignored. Brackets indicate optional items; if the brackets are followed by "`+`", the item must appear one or more times; if they are followed by "`*`", the item may appear zero or more times.

We have already seen a sample schema file in Figure 1. The `alias` and `verbosename` keywords may be used by programs that present the relation to the user, but they have no impact on the contents of the relation files. The `hashsize` keyword, if present, determines the number of hash buckets used during stabilization. The other options are discussed in subsequent sections.

2.2 Tuple syntax

Tuples in the relation are also stored in Ascii and may be entered by an ordinary text editor. The storage format is readable with some effort, but users typically deal with the presentation format of the data. The `qddb` package includes routines to convert between presentation and storage format.

Each non-empty attribute in the tuple is presented as an attribute name followed by its value. Expandable attributes are presented once for each instance. The BNF grammar for the presentation format of tuples is shown in Figure 3.

Figure 4 shows the presentation format of a tuple conforming to the schema in Figure 1.

```
UniversityID
Name (
        FirstName
        MiddleName
        LastName
)
Address (
        Street (
                Number
                Road
                Apartment
        )
        City
        State
        Zip
        Country
        Phone*
        EMail
)*
Course (
        Number
        Semester
        Year
        Grade
        Credits
)*
```

Figure 1: A sample schema for student records

```
Schema ::= [ Options ] [ Record ]+
Options ::= [ HashSize ] [ CacheSize ] [ CachedHashing ]
                [ ReducedID ]
HashSize ::= 'hashsize' '=' Integer
CacheSize ::= 'cachesize' '=' Integer
CachedHashing ::= 'Use' 'Cached' 'Hashing'
ReducedID ::= 'Use' 'Reduced' 'Attribute' 'Identifiers'
Record ::= [ IRecord ]+
IRecord ::= ID [ '(' Record ')' ]* [ '*' ]
ID ::= String [ 'alias' String ]* [ 'verbosename' CString ]
String ::= [a-zA-Z][a-zA-Z0-9]*
CString ::= '"' [ AnyPrintableASCIICharacter ]* '"'
Integer ::= [0-9]+
AnyPrintableASCIICharacter ::=
                [A-Za-z0-9!@#$%^&*()+-=|\,./<>?~`{}[]␣_]
```

Figure 2: BNF for the format of a Schema file

```
FullTuple ::= '$NUMBER$' '=' '"' [0-9]+ '"' ';' Tuple
Tuple ::= [ Record ]*
Record ::= [ IRecord ]+
IRecord ::= ID | String [ '(' Record ')' ]+
ID ::= String [ '=' CString ]
String ::= [a-zA-Z]+[a-zA-Z0-9]*
CString ::= '"' [ AnyPrintableASCIICharacter ]* '"'
AnyPrintableASCIICharacter ::=
                [A-Za-z0-9!@#$%^&*()+-=|\,./<>?~`{}[]␣_]
```

Figure 3: BNF grammar for the presentation format of tuples

```
$NUMBER$ = "0";
Name (
        FirstName = "Joe"
        LastName = "Student"
)
Address (
        Street (
                Number = "456"
                Road = "Somewhere Lane"
        )
        City = "Little Town"
        State = "AZ"
        ZipCode = "67890-1234"
)
Address (
        Street (
                Number = "123"
                Road = "Zimbob Lane"
                Apartment = "3A"
        )
        City = "Lexington"
        State = "KY"
        ZipCode = "12345-1234"
)
```

Figure 4: An example of a tuple in presentation format

3 File structure

Each `qddb` relation is implemented as a Unix directory containing several files and subdirectories. The name of the relation is the same as the name of the directory that contains these files. All files are in Ascii. Some files are **structure** files, which assist in performing searches. Structure files generally point into other files by means of **locators**, which are (*offset*, *length*) integer pairs. The files are as follows.

1. `Schema` – The schema for the relation, following the syntax given in Figure 2 above.
2. `Stable` – A file containing all stable tuples in the relation. They are represented in storage format, in which tuples are separated by blank lines, and every attribute starts on a new line with an identifier, which is an (attribute identifier, instance number) pair. Nested attributes include another such pair for each level of nesting. For example, the apartment number of a student's second address has the identifier "%3.2.1.1.3.1", meaning that the attribute is the second instance of the third attribute given by the schema in Figure 1. The meaning of this particular attribute identifier can be summarized as follows:
   - 3.2 = Address: second instance
   - 1.1 = Street: first instance
   - 3.1 = Apartment: first instance

   Each tuple has an implicitly defined attribute called `$NUMBER$` used as a unique identifier for that tuple in the relation. Identifier "%0" stores both the unique identifier and a validity flag (either "V" or "I"). The whole tuple can be marked invalid (during an update, for example) by changing this flag; since `Stable` has the same length afterward, all locators pointing into it are still usable.
3. `Index` – A structure file containing all hash buckets for the hash table. Each bucket contains zero or more entries. Each entry contains a key followed by a list of locators for that key into `Stable`.
4. `HashTable` – A structure file containing the hash table. Each entry contains a locator into `Index`, the number of keys that hash to this entry, and the value of the hash function for this entry (so sparse tables are efficiently represented). The number of buckets, hence the size of the hash table, is computed at stabilization time to keep the number of collisions reasonably small without making the table excessively long. The default hash size may be overridden by the `hashsize` option in the `Schema`.
5. `Stable.key` – A structure file containing a complete list of locators for the tuples in `Stable`. This file is useful at stabilization time and whenever a program needs to read all tuples in the relation.
6. `Changes` – A directory containing a file for each tuple that has been updated since the relation was last stabilized. Each such file is named by the serial number of the affected tuple and contains that tuple in storage format. The affected tuples are marked invalid in `Stable`.
7. `Additions` – A directory containing a file for each tuple that has been added since the relation was last stabilized. Each such file is named by the serial number of the affected tuple and contains that tuple in storage format.
8. `Additions/NextEntry` – A file containing the next serial number to be used when a tuple is added.

Typically, programs that query the relation initialize themselves by bringing `HashTable` into memory. Thereafter, a query on a single key requires (1) computing the key's hash value, (2) examining the hash table to find the locator into `Index` for the associated bucket, (3) reading a contiguous section of `Index` to obtain locators into `Stable` for the tuples containing that key, and (4) reading a contiguous section of `Stable` for each tuple to be presented. The list of locators built in step (3) is stored in an internal form called a `TupleList` (see Section 4), which can be manipulated before step (4), for example, to remove unwanted entries. Thus, the solution set X of a simple query on stable data can be obtained in $|X| + 2$ *read* system calls, including the one needed to bring the hash table into memory.

Each query must also search instable data. The additional cost is one extra *open/read* pair plus one read of the `Changes` or `Additions` directories for each changed or added tuple. Thus, the number of excess *open/read* system calls is directly proportional to the number of added and updated tuples in the relation, that is, the number of tuples in the instable part.

In addition to the extra system calls, instable relations also require significant computation, because a string-search routine like `fgrep` must be used to determine the presence of keys.

We have considered an alternative implementation: to log all updates in a single `Log` file, with locators to that file stored in a separate structure file called `LogLocators`. Each update would require invalidating the appropriate entry in `LogLocators` and appending the update to `Log`. This representation has the advantage that there are fewer *open* calls needed to search updated tuples. (*Open* calls are potentially expensive.) The log file approach is therefore more efficient than our current method for querying instable parts of a highly modified relation.

The log file representation has the disadvantage that it requires more computation to perform an update. Also, simultaneous updates to different tuples require record locking in potentially many portions of the two files. Our current approach has the advantage of less complicated storage management and the expense of only a single pair of open/write calls per update. Opening a single file for exclusive use is sufficient to prevent race conditions and allows all tuples to be updated at once.

We accept the performance penalty of our method for instable relations because we assume that relations are changed infrequently; however, we believe that log files would be a useful addition to `qddb`.

4 TupleLists

A `TupleList` is an internal data structure (a singly-linked list of locators into `Stable`) used by programs to represent sets of tuples. Search routines generate a `TupleList` for each single-key query. Multiple-key queries generate multiple `TupleLists` that are merged in a manner consistent with the type of query. Library routines accept, manipulate, and return these lists instead of dealing with tuples directly. The values of the tuples can be extracted by following the locators.

More precisely, a `TupleList` node contains the following information:

- The unique serial number of the tuple
- A locator for the tuple in `Stable` (if applicable)
- The type of the tuple: `original` for tuples in `Stable`, `change` for tuples in the `Changes`

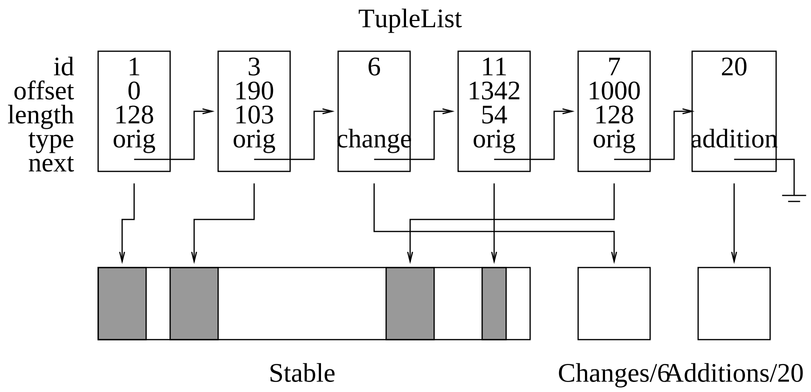

Figure 5: TupleLists and a `qddb` relation

directory, and `addition` for tuples in the `Additions` directory

Attribute values are not directly described by `TupleLists`. Figure 5 shows the relationship between a `TupleList` and the various `qddb` files.

`TupleLists` allow us to prune the solution set of a search before any tuples are read from the relation. The binary operations available on `TupleLists` are set union, intersection, and exclusion. We perform these operations by traversing the two `TupleLists` in the appropriate manner. We keep these lists sorted by serial number to make the cost of all operations linear in the combined size of the two `TupleLists`.

For example, we might want to find a particular paper written by several people in a bibliography relation. The intersection of the `TupleLists` that are produced by the individual searches for each person produces a reasonable list.

5 Enhancements to basic `qddb`

The straightforward ideas presented so far constitute **basic `qddb`**. In this section we describe enhancements to basic `qddb`, which generally involve creating optional new structure files at stabilization time in order to make certain types of access more efficient. If these structure files are missing, we disallow any query that would require them. Most of these enhancements were motivated by our `SQL`like language, `QSQL`, to which we return in Section 6.

5.1 Attribute-specific queries

It is traditional in languages like `SQL` to base queries on characteristics of particular attributes, not of tuples as a whole. To add such a facility, we could join the attribute name to each word for the purpose of hashing. Unfortunately, it would no longer be easy to perform attribute-independent queries, and both `HashTable` and `Index` would grow.

We chose a less restrictive method for attribute-specific queries based on an extra structure file called `AttributedIndex`. For each key, `AttributedIndex` augments each locator with an identifier that describes the attribute in which the key occurs. Attributes are designated by **enumerated**

attribute identifiers, which are single integers that provide a way to uniquely identify each attribute name in the `Schema`. This information is readily available at stabilization time and adds little overhead to the stabilization process. The corresponding `AttributedTupleList` data structure also decorates each entry by an enumerated attribute identifier.

There is no extra system call overhead involved with this method. The only performance penalties are: (1) space: `AttributedIndex` is slightly larger than `Index` to accommodate reduced attribute identifiers and some new entries (if the same word appears in several attributes of the same tuple), and (2) time: Uninteresting nodes must be deleted from any `AttributedTupleList` before tuples are read from the relation.

Depending on the expected queries, relations may be stabilized with either or both of `Index` and `AttributedIndex`.

5.2 Regular expression searching

Basic `qddb` searches on keys. Instead, an SQL query might be posed with respect to a regular expression to be matched directly against attribute values. For example, searching for the pattern "`\(606\)-293-([0-9][0-9][0-9][0-9])`" should produce a list of telephone numbers in a particular city zone.

Regular expression searching usually requires a linear search of all text in the relation. Some regular expressions are simple enough that they are essentially equivalent to an intersection of ordinary word queries. For example, the regular expression "operating systems" is almost equivalent to the intersection of the queries "operating" and "systems", although the intersection will include tuples that contain these words in nonadjacent or reversed positions.

Reducing the regular expression to ordinary queries is often inadequate. Regular expressions cannot make use of hash tables, so we need another way to narrow the search. It might seem reasonable to search linearly through `Index`, but that file is often longer than `Stable`, since it contains lists of locators.

Instead we rely on a new optional structure file, `KeyIndex`, consisting of a single locator into the `Index` file for each key. (See figure 6.) `KeyIndex` is meant to be searched linearly. Regular expressions to be matched within a single key are matched against all the keys in the `KeyIndex` file. However, regular expressions that are meant to cross key boundaries (or involve non-key punctuation) must still be applied to the `Stable` file and any instable parts of the relation. We have observed that the size of `KeyIndex` is only about 15-35% the size of `Index` and about 20-30% the size of `Stable`, so the time savings are significant for appropriate regular-expression queries.

The expense of this method is two *open* system calls and

$$\frac{\text{sizeof(KeyIndex)}}{\text{sizeof(pieces read)}} + \text{(number of entries found)}$$

read system calls. An extra read of `Index` is required for each key found. When it is possible to read the entire list of keys into memory, the computation required for regular expression searching will be proportional to the number of keys in the relation, and the number of reads will be one plus the number of entries found. The amount of computation required for regular expression searches is, of course, large.

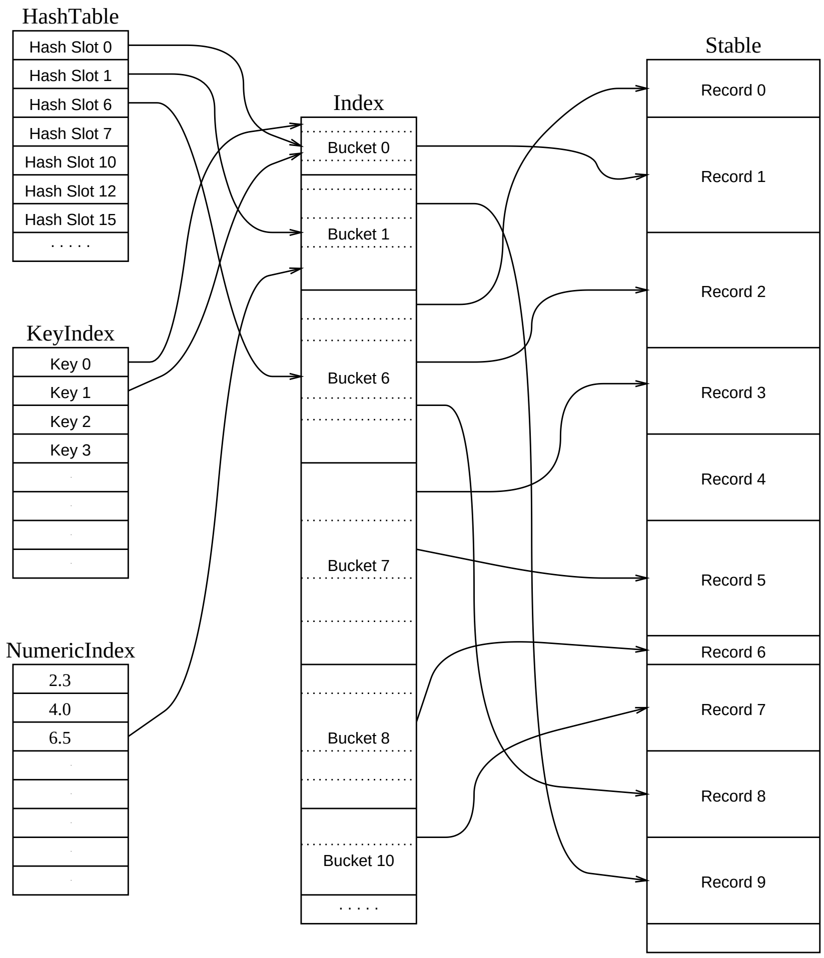

Figure 6: `Qddb` Files

5.3 Range searching

`SQL` allows queries that treat attributes as numbers for comparison testing. In order to handle such queries, we introduce another optional structure file, `NumericIndex`, which contains a sorted list of numeric keys (represented in Ascii, decimal fixed point), each with a locator into `Index`. (See figure 6.) A query program can then perform a binary search on `NumericIndex` to construct a `TupleList` matching any numerical range or set of ranges. This binary search may read all of `NumericIndex` and perform an internal search or may probe it as an external data structure until the region of interest is small enough to read into memory. Similarly, we use `KeyIndex` (defined above), which we sort alphanumerically, for range searches of arbitrary strings.

5.4 Direct access hash tables

Large relations tend to have large hash tables. The space problem is more severe when we run an `SQL` query that accesses several relations at once. We wish to keep the memory requirements of query programs as small as possible so that small machines can be used for queries, even though larger ones might be needed for stabilization.

We therefore allow an optional replacement for the `HashTable` structure file called `RandomHashTable`. This table contains the same information as `HashTable`, but each entry has a fixed length, empty entries are not omitted, and we omit recording each hash-bucket number. When this table is present, query programs access individual hash table entries on demand without reading and saving the entire table. They may cache entries if they wish.

The time penalty that we incur is an occasional single read of the `RandomHashTable` file or a search through the cashed entries on each query. The space penalty is that `RandomHashTable` is generally considerably larger than `HashTable` because it does not suppress empty entries and because each entry must be lengthy enough to hold the largest entry.

5.5 Encoding `Stable`

The values in `Stable` tend to occupy only a part of the file. The attribute and instance identifiers (such as %3.2.1.1.3.1) sometimes require more storage than the actual data. In one genealogical database we have designed, more than 40% of the `Stable` file is occupied by non-value information.

Since the identifiers are usually very repetitious, we can abbreviate them for storage efficiency. The association between abbreviations and full identifiers is stored in an optional `Abbreviations` file. The set of abbreviations needed is discovered at stabilization time. The `Stable` file only uses these abbreviations, leading to a significant potential space savings, especially in large relations with deeply nested attribute structures. Entries in the `Changes` and `Additions` directories may use abbreviations where appropriate, or full identifiers where desired.

The performance penalty, which involves reading `Abbreviations` once and then performing constant-time array lookup, is incurred only by those programs that deal with presentation format. The identifiers are not necessary for storage-format programs.

6 Support for `QSQL`

Implementing `QSQL` required that we define the meaning of standard database operations in the presense of missing, expandable, and structured attributes.

Missing attributes can be either applicable or inapplicable [Cod90]. An applicable missing attribute is one that should have a value, but the value was unknown at the time of entry. An inapplicable missing attribute is one that has no value and the attribute does not have meaning for a particular tuple. `QSQL` treats all missing attributes as inapplicable. When necessary, a user can put a “NULL” value into an attribute to assert that it is applicable.

`QSQL` views tuples with expandable attributes as several tuples that agree on their simple attributes and contain all combinations of values of the expandable attributes.

Subattributes must be grouped together or chaos will result, as seen in the subattribute grouping from the student database shown in Figure 1. If each street is not grouped with the proper city, the address is nonsense. We must not separate the subattributes when translating tuples to a relational format. `QSQL` extends standard `SQL` by providing a syntax for specifying subattributes. In addition to traditional constructs of the form $P.attribute$ = *"string"*, `QSQL` also allows constructs of the form

$P.attribute.subattribute = "string"$.

Although QSQL methods for handling expandable and structured attributes may appear to cause update anomalies, they do not. Decomposition of a relation into a normal form is not necessary, because a tuple containing expandable and structured attributes is still a single tuple, albeit *viewed* as multiple tuples. Updates only operate on a single physical tuple (multiple logical tuples). Consistency can be checked at a higher level.

We chose $UNIX^{tm}$-style regular expressions because they are much more flexible than those defined by the standard. Standard SQL allows only wildcard replacement.

7 Current status

Our first implementation of qddb, which only allowed Boolean combinations of single-key queries, contained about 1000 lines of C code and was programmed by one person over the course of about a month. The name qddb stands for "quick and dirty", because our original intent was to provide a simple database package that would not require much effort to construct but would be useful to a wide community. Since then, we have expanded, enhanced, and (hopefully) improved qddb. Qddb now contains over 10,000 lines of C code, much of it devoted to QSQL. The QSQL interpreter is almost complete, and we hope to add embedded QSQL in the future. At present, there are few users, but the suite of programs has not been publicized, nor has various helpful software been written to assist the user.

Qddb meets its goals of portability and speed. We access relations from hosts of five different hardware architectures (Sequent symmetry, Sun 3, Sun 4, Vax, and DecStation) with identical code across the network file system (NFS) on different versions of BSD Unix. The same code also runs on various System V machines (AT&T 386, 3B1, 3B2).

Qddb is relatively fast. It takes an average of 5 minutes on a 16 MHz 80386 machine to stabilize a relation of length 3,138,793 characters (24,481 keys). Single-key queries take only 35–300 milliseconds to return a TupleList with 0–30 tuples on a stable relation using HashTable, not RandomHashTable. Reading and parsing a 20,000 entry sparse HashTable (mostly full) consumes less than 6 seconds. Reading and parsing a single hash table entry takes about 10–20 milliseconds on the average (used for reading the hash table on demand, possibly caching entries). Individual tuples can be read in 0.6–40 milliseconds, depending upon whether the kernel has cached the relevant portion of the Stable file and on the location of the tuple in the file. The most time-consuming single-key query that we can produce completes in under 800 milliseconds of real time (returning 30 tuples). TupleList intersection using two or more keys consumes well under 800 milliseconds, since the number of tuples in the solution set is decreased by intersection. The time for TupleList union is slightly less than linear in the number of keys.

We don't have many utilities yet. The following are fairly straightforward tools that would be worthwhile.

- Presentation utilities that present subsets of a tuple, eliding parts that the user's view wishes to exclude.
- A tool for building user interfaces and pretty printers (report generators) for individual relations.

8 Potential extensions

In this section we discuss ideas that might make `qddb` more suitable to a wider range of applications. We do not contemplate implementing them at this point.

8.1 Generalization of keys and hashing

A tunable definition of a key would be a useful way to trim `Index` to a more reasonable length. A key-definition language could be supplemented by a list of specific words to exclude from `Index`. The query processor could warn users when queries contain invalid keys.

We have chosen to use delimited words as keys; this choice colors what queries are simple and which are complex. We could use suffix arrays [MM90] instead of hashing in order to allow a quick query for any substring. This data structure requires about four times as much storage as `Stable`, but provides $O(\log n)$ search time for arbitrary strings.

8.2 Large relations, frequent updates

Our approach is not appropriate for large relations that are heavily updated. The penalties incurred by search increase linearly with each update. However, stabilization is too expensive to undertake frequently in a large relation.

We will present two approaches to this dilemma. The first partially stabilizes after every few changes. The other incrementally stabilizes after every change.

Each **partial stabilization** is a group of `Stable` and structure files, each group containing a subset of the total relation. Any query needs to search all the groups, so we would prefer to limit how many there are. However, such a limit implies that the groups have many members, making them expensive to create.

One policy for maintaining groups is to merge groups when there are too many. Let n be a threshold value, say 100. We will allow at most one group of size $2^i n$ for any $i \geq 0$. As soon as there are n instable tuples, they are stabilized into a group by themselves. Such a stabilization may lead to two groups of size n. In general, if there are already groups of size $2^i n$ for $0 \leq i < j$, but not for $i = j$, all are merged with the new group and become one new stable group of size $2^j n$.

Incremental stabilization is an alternative approach that aims at reducing the costs involved in searching a updated relation. These costs are

1. The `Additions` and `Changes` directories must be opened and sequentially read to find the entries that have been added or updated since the last stabilization.
2. Each addition or change entry is contained in its own file. Each search must open and read the contents of all such entries.
3. The contents of each addition or change must be parsed into keys during the search.

We can greatly reduce these costs by performing intermediate calculations during each update. We maintain secondary structure files that pertain to changes and additions: `Index.Change`, `HashTable.Change`, `Index.Addition`, and `HashTable.Addition`, identical in function to their stable counterparts. The optional structure files could also be included. The locators in the new `Index` files contain only a file number (same as the file name in the `Changes` and `Additions` directories), that is, the serial number of the updated tuple. The file number is used instead of an offset in the locator because it is the file name and not the offset in `Stable` that must be used to retrieve the data.

Each search performs the following operations: (1) search via the primary structure files, (2) search via the secondary files, (3) ignore stable `TupleList` nodes that have the same identifier as a changed node (we use the offset in `Stable` as a unique identifier and use that identifier to name the files that contain changed entries).

Each search suffers two additional open/read system-call pairs (one for additions and one for changes) with an additional *open/read* system call pair for each entry that matches the query. This method totally eliminates cost 1 and greatly reduces the computation necessary for costs 2 and 3.

An update to the relation is performed by deleting all previous references to the updated entry in all secondary `Index` files (to avoid inaccurate locators) and appending a new locator to all appropriate entries in the correct secondary `Index`. An updated entry is marked invalid in `Stable`. The cost of this approach is quite reasonable when the number of change and addition entries is relatively small. As the number of updates increases, the cost of performing an update increases due to the increasing size of the corresponding `Index` file.

8.3 Transaction support

We do not currently support transactions in `qddb`. Failure atomicity could be implemented by giving each transaction private supplemental `Changes`, `Additions`, and `Deletions` directories. These directories would contain only those tuples that are updated from the standard directories. The `Stable` file would not be updated during a transaction, but entries in the supplemental `Deletions` or `Changes` directories would override entries in `Stable`. The ordinary `Changes` and `Additions` directories would maintain their usual meaning.

Aborting a transaction only requires discarding the private directories. Committing a transaction requires merging private changes, additions, and deletions into the ordinary files. New changes replace conflicting old ones. Where there is no conflict, the new change invalidates the corresponding entry in `Stable`. New additions are renamed if necessary to give them unique numbers. Deletions invalidate corresponding entries in `Stable`.

`Qddb` currently uses record locking to insure that individual tuples remain consistent. Implementation of appropriate locking strategies would be necessary to guarantee serializable transactions.

8.4 Distributing `qddb` by replication

`Qddb` can be easily enhanced to distribute replicated data. Given the structures outlined above for failure atomicity, we can transfer updates performed on one site to the other sites by sending the `Changes`, `Additions`, and `Deletions` directories. Restabilization can occur on each site independently and concurrently, so there is no need for a central site to transfer the entire relation at any time.

8.5 Default values

We could allow the specification of default values in the `Schema` file for applicable attributes. Default values are useful when a particular attribute has a value that occurs frequently. For example, the *Country* attribute in the student database of Figure 1 might usually have the value *USA*, which would be a reasonable default.

8.6 Consistency constraints

If the relation designer wishes each tuple to satisfy some consistency constraint, we could easily provide a hook for checking after each update. However, most such consistency checking can be layered on our library routines. We do not currently plan to build any consistency checking utilities or routines.

8.7 Attribute functions

We could allow some attributes to hold values not stored in `Stable` but computed on demand from the values of other attributes. For example, in our student database of Figure 1, the value of a *MailingLabel* attribute could be computed from the *Address* attribute, and *GradePointAverage* could be computed from the *Course* attribute. Again, hooks could be provided for the programmer to provide these routines.

9 Comparisons to commonly used approaches

Our design of `qddb` was influenced strongly by the *refer* suite of programs [Les78], which store and retrieve bibliographic data. Like *refer*, `qddb` uses Ascii for the principal data file, allows arbitrary strings as attribute values, allows attribute values to be missing, builds index files for quick searches, and searches for keys independently of which attribute they are in. Unlike *refer*, `qddb` permits a relation to contain both a stable and an instable component, permits subattributes, and distinguishes Ascii presentation from Ascii storage format.

We believe the flexibility of our approach is its major advantage over many other approaches. Most techniques for retrieval are optimized for access using keys in particular attributes. We do not think that restricting to given attributes is always reasonable; some applications may require access to entries containing a specific key in an arbitrary attribute. Many other methods require that a key be unique in the relation. We feel that this assumption is too restrictive. We have shown how to enhance our approach to allow attribute-specific retrievals, so our approach may be used whenever the other approaches are appropriate. Consistency checkers can police attribute uniqueness when that restriction is useful.

Our implementation imposes a particular structure on the relation file. Other methods, such as B-trees [Bay72], impose a rigid structure requiring a number of *read* system calls that increases logarithmically (with a possibly large base) with the number of keys in the relation. The number of reads required by our method does not increase with the size of the relation, although the underlying file organization of Unix will impose an increased number of low-level disk reads for random access within very large files. A truly native implementation of `qddb` would place raw disk addresses in locators and avoid this extra level of overhead [Sto81].

9.1 Indexed files

Methods that maintain indexed files, such as the indexed-sequential method, keep an index that consists of (*key*, *index number*) pairs. Generally, index files are kept in a tree form so that a particular key may be found in logarithmic time. The entire index is sometimes kept in memory to make the search fast; but this step is not possible on machines with limited memory or with very large relations. An index must be maintained for each attribute on which searches are allowed.

`Qddb` only requires that the hash table and a single hash bucket reside in memory at any given time. Using `RandomHashTable`, we reduce the memory requirement to one hash-table entry and one hash bucket. The hash buckets are read at the time of the search and discarded afterwards to free memory. `Qddb` requires enough memory to hold the largest hash bucket, so a good hash function will keep the memory requirements small. The support of `TupleList` operations requires enough memory to hold a `TupleList` for each key used in the operations; a query using a dozen keys would require memory for a dozen `TupleLists`. A `TupleList` requires memory proportional to the number of entries in the solution set of the single key query.

9.2 Hashed files

A classical hashed-file scheme builds a separate hash file for each attribute on which searches might be made[Wie87]. Within such a file, each hash bucket contains the contents of (not locators to) all tuples that have keys in a given attribute that hash to that bucket. Hashed files provide reasonable search times when:

1. The hash function is good (the number of collisions is very small).
2. Only searches on keys within a single attribute are required.
3. The size of the individual entries is reasonably small.

The disadvantages of hashed database files include an extra read of an entry for each collision and the inflexibility of hashing keys in a single attribute.

`Qddb` uses the method of hashed files in its indexes to produce the speed we desire. Since we don't update the hash table until the next stabilization, we need no extra space for future additions to buckets, and the hash buckets are contiguous and of known length. We can always read an entire bucket with one system call.

9.3 Tree-structured files

Tree-structured files provide a tree structure (usually binary) for a particular attribute in each entry. Multi-attribute searches require a separate tree for each attribute on which queries are allowed. The number of reads necessary for a single attribute query is logarithmic. Multi-attribute searches require a search for each attribute, so the number of reads is:

$$number\ of\ attributes \times \log_2(number\ of\ entries)$$

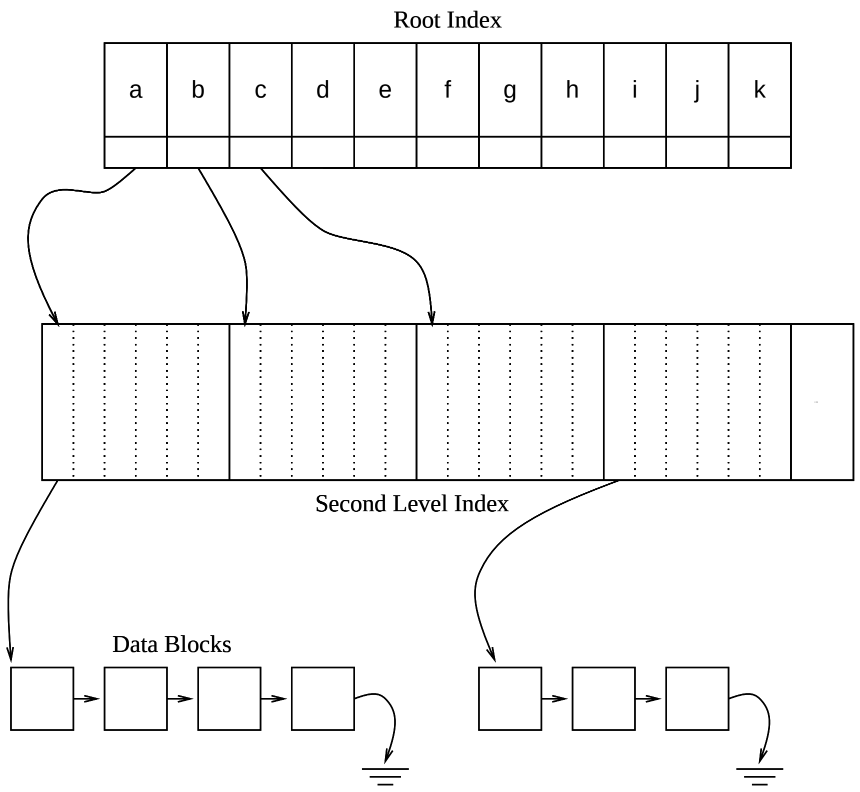

Figure 7: A two-level b-tree

This estimate is based on a balanced binary tree.

Our approach also requires a search for each attribute in a multi-attribute search, but the number of reads required for each search is a constant (two) plus the number of entries that satisfy the query. The number of trees required to allow a search on any attribute in a tree-structured database is the number of attributes, so each attribute in each entry must have two pointers associated with it. Any search that is not specific to a particular attribute would require searching for the key in every tree. Our approach requires a similar amount of storage, but provides a significant performance improvement for large relations and searches that are not specific to a particular attribute.

9.4 B-trees

B-tree implementations generally impose a fixed-length key so that blocks of keys are of a fixed size (see Figure 7). Our approach is somewhat similar to the following variant of a two-level B-tree: We could fix the size of the each bucket so that the size of each read of `Index` is of a given size ($\pm$ some percentage). The entire `Index` file is sorted by key. Instead of a hash table, we could access the `Index` file via the top-level node, which would be a list of key ranges and associated locators to buckets. A binary search on the top-level node produces a locator to the bucket containing the desired key, where another binary search finds the appropriate locators to `Stable`.

The number of *read* system calls is identical to the number required by our approach, but it allows range searching on Ascii keys. Another advantage to B-trees is that the size of the largest bucket is somewhat constrained. We do not have control over the size of the `TupleList` associated with any given key, but our stabilization process could easily find a good distribution. The b-tree approach costs somewhat more than our hash-table approach in storage (keys are arbitrary lengths) and in memory lookup (a logarithmic number of string comparisons must be made).

10 Discussion

We have shown that a large performance increase over general databases can be achieved when managing relatively stable data. Our approach allows a very flexible database format yet provides performance that is as good or better than other databases that have inflexible formats. Our Ascii format allows portability between architectures, programmer readability, and a degree of error tolerance. The error tolerance can be important if a particular disk error cannot be corrected by the underlying operating system. A bad sector in our relation files does not affect the entire relation, and the relation can be manually corrected by an experienced user. `TupleLists` allow the entire solution set of a query to be found without reading any tuples from the relation.

We have detailed enhancements to the structure files that allow fairly efficient regular-expression and numeric-range searches and limited update without severely affecting search performance. Database administrators may choose to build optional structure files at stabilization time to provide only those features that enhance performance for a particular application. These options are currently specified in `Scheme`.

We have compared our approach to the approaches commonly used in generic database design. We purposely disregarded transaction management details such as file and record locking during update, since we feel that such necessities are obvious. The fact that the instable parts of a relation are segregated may make rollback particularly easy in `qddb`. Increasing version numbers may be appended to the file names of old updates to facilitate rollback of multiple updates.

We suggest that a specific implementation of our approach should stabilize the relation on a regular basis; but of course, the interval should depend upon the specific application. We foresee that some large relations may need stabilization only a few times per year, whereas others may need stabilization on a nightly basis. A consequence of our current implementation is that the relation must be inactive during a stabilization. We might be able to reduce the time a relation is unavailable by allowing new Additions and Changes directories to be built during stabilization.

The major disadvantages of our approach are threefold: performance degradation on updated relations, the computation needed for stabilization and the attendant unavailability of the data, and the storage required to hold the structure files. We accept the performance penalties for updated relations, since our initial assumption was that we are dealing with relations that are rarely updated. The computation required to stabilize the relation cannot be avoided; the temporary unavailability of the data, however, may make our approach unacceptable to applications that require high availability. We have noticed the storage required for our `Index` files is usually at least as large as the relation, and in many cases, larger. Much of this size is attributable to our definition of a key; some words such as "Road" (for the student relation) or "the" (for a bibliography/abstract relation) appear many times and produce large `Index` files. A more restrictive definition of a key would significantly decrease the size of the file. Even with large relations, the storage cost should not be a prohibitive factor, since the cost of large disk drives is rapidly decreasing. The stable part of the relation could even be stored on write-once optical disks, with records indicating which stable tuples are invalid kept on ordinary writable store.

We have observed that our methods perform equally well regardless of the size of the relation. A relation with 1,000 entries requires roughly the same amount of time to satisfy a query as a relation with 1,000,000 entries. The major difference in speed is the startup time for the query program to read `HashTable`. `RandomHashTable` seems to virtually eliminate this difference. Query times do not increase noticeably when multiple-attribute searches are performed. We suggest that our methods are quite suitable for PC class machines or slow optical disks.

11 Packaging

`Qddb` is available as a set of utility programs and a library of routines that are useful for building other programs. This code and documentation is available by anonymous ftp from `ftp.ms.uky.edu:pub/unix/qddb/`. A listing for the *qs* program is given in Figure 8 to demonstrate how utility programs are easily built upon the library routines.

Some of the programs in the suite are:

- *newdb* – Create an empty relation and build a schema file.
- *qadd* – Add a tuple to the specified relation.
- *qconv* – Convert a relation in presentation format to `qddb` internal format.
- *qedit* – Search a `qddb` database for a Boolean combination of single-key queries and edit or list all entries found.
- *qindex* – Rebuild structure files, such as `HashTable`, `Index`, `KeyIndex`, and `NumericIndex`. The current implementation requires that all keys fit into memory. On small machines, such as a PC, this restriction is unacceptable. We plan to build a second version of *qindex* that uses temporary files instead of memory; however, the stabilization process will be considerably slower. We recommend that stabilization occur on large machines when they are available. The relation files can then be transferred to a PC for query sessions, since our format is machine-independent.
- *qkeys* – Rebuild `Stable.keys`.
- *qs* – Perform interactive queries; initializes by reading `HashTable`.
- *qstab* – Merge all changes and additions into `Stable`.
- *qschemamod* – Allows arbitrary update of `Schema`.
- *qsql* – Interactively process `QSQL` queries and updates.

The principal library routines are:

- `void InitHash()` – Initialize the internal hash table.
- `void MakeHash(char *RelationName)` – Read `HashTable` for the given relation.
- `void ChangeEntry(char *RelationName, Entry TheEntry)` – Make a change to a tuple by invalidating the old entry and adding the new one to the appropriate location.
- `void AddEntry(char *RelationName, Entry TheEntry)` – Add a new tuple to the relation.

```
void main(argc, argv)
    int                 argc;
    char                *argv[];
{
    Schema              *schema;
    char                String[BUFSIZ];
    char                Relation[BUFSIZ], *rel;
    KeyList             *list = NULL;
    size_t              Length;
    Qddb_SearchArg      search_args;

    Qddb_Init();
    rel = Qddb_FindRelation(argv[1]);
    strcpy(Relation, rel);
    schema = Qddb_InitSchema(Relation);
    while (fgets(String, sizeof(String), stdin) != NULL) {
    char        *ptr;
    KeyList        *LastList = NULL;
    ptr = String;
    while (*ptr != '\0') {
        char    *token;

        token = ptr;
        while (!isspace(*ptr) && *ptr != '\0')
        ptr++;
        if (*ptr != '\0') {
            *ptr++ = '\0';
            while (isspace(*ptr))
                ptr++;
        }
        search_args.Alpha = token;
        search_args.Type  = ALPHA;
        list = Qddb_Search(schema, &search_args, &Length, -1);
        if (LastList != NULL)
            list = LastList =
                Qddb_KeyListOp(schema, list, LastList,
                      QDDB_KEYLIST_OP_INTERSECTION,
                      QDDB_KEYLIST_FLAG_DELDUP_SAMEENTRY);
        else
            LastList = list;
    }
    if ((argc == 3 && strcmp(argv[2],"-p") == 0) || (argc == 2))
        PrintKeyList(schema, list, True);
    else if (argc == 3 && strcmp(argv[2],"-n") == 0)
        PrintKeyList(schema, list, False);
    }
    exit(0);
}

void PrintKeyList(schema, list, Print)
    Schema          *schema;
    KeyList         *list;
    Boolean         Print;
{
    char            *Buffer = Malloc(BUFSIZ), Valid;
    int             DBFile;
    unsigned        len = BUFSIZ;

    DBFile = OpenDatabase(schema->RelationName, 0);
    while (list != NULL) {
    switch(QDDB_KEYLIST_TYPE(list)) {
    case ORIGINAL:
        if (list->Length > len) {
            len = list->Length;
            free(Buffer);
            Buffer = Malloc(len);
        }
        lseek(DBFile, list->Start, 0);
        Read(DBFile, Buffer, list->Length);
        sscanf(Buffer, "%%0 %c", &Valid);
        if (Valid == 'I')
            break;
        if (Print == True)
            Write(1, Buffer, list->Length);
        break;
    case CHANGE:
        if (Print == False)
            break;
        sprintf(Buffer, "cat %s/Changes/%d", schema->RelationName, (int)list->Number);
        system(Buffer);
        break;
    case ADDITION:
        if (Print == False)
            break;
        sprintf(Buffer, "cat %s/Additions/%d", schema->RelationName,
            (int)list->Number);
        system(Buffer);
        break;
    default:
        PANIC("BAD TYPE IN KeyList");
    }
    if (Print == True)
        write(1, "\n", 1);
        list = list->next;
    }
    free(Buffer);
    close(DBFile);
}
```

Figure 8: C Listing for *qs*

- `Schema *ReadRelationSchema(char *RelationName)` – Read the `Schema` file for the relation, parse it and return an internally represented schema.
- `TupleList *Search(char *RelationName, char *SearchString, unsigned long *NumberOfEntriesRead)` – Search the relation for a particular string and return the `TupleList` and the number of entries found.
- `TupleList *TupleListUnion(TupleList *TupleList1, TupleList *TupleList2)` – Perform the set union on the given `TupleLists` and return the result. The given `TupleLists` are destroyed upon return from this routine.
- `TupleList *TupleListIntersection(TupleList *TupleList1, TupleList *TupleList2)` – Perform the set intersection on the given `TupleLists` and return the result. The given `TupleLists` are destroyed upon return from this routine.
- `TupleList *TupleListNot(TupleList *In, TupleList *ButNotIn)` – Perform the set exclusion on the given `TupleLists`; return all nodes in *In* but not in *ButNotIn*. The given `TupleLists` are destroyed upon return from this routine.
- `unsigned long SizeOfFile(int FileDesc)` – Return the size of the given file; usually used to read the contents of a file in a single read.
- `void ReadEntry(int FileDesc, Entry ThisEntry, unsigned long Start, unsigned long Length)` – Read an entry with the given offset and length in the file.
- `void WriteEntry(int FileDesc, Entry ThisEntry)` – Write the entry into the file at the current location.
- `unsigned long GetEntryNumber(Entry TheEntry)` – Return the unique identifier of the entry.
- `Boolean EntryInvalid(Entry TheEntry)` – Return True if the entry has been updated; false otherwise.

References

[Bay72] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[Cod90] E. F. Codd. *The Relational Model for Database Management, Version 2.* Addison Wesley, 1990. ISBN 0-201-14192-2.

[Dat87] C. J. Date. *A Guide to The SQL Standard.* Addison-Wesley, 1987. ISBN 0-201-05777-8.

[KS86] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* Computer Science Series. McGraw-Hill, 1986. ISBN 0-07-044752-7.

[Les78] M. E. Lesk. Some applications of inverted indexes on the unix system. *Unix Programmer's Manual*, 2b, 1978.

[MM90] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *Proceedings of the first Symposium on Discrete Algorithms*, January, 1990.

[Sto81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7), 1981.

[Wie87] Gio Wiederhold. *File Organization for Database Design.* Computer Science Series. McGraw-Hill, 1987. ISBN 0-07-070133-4.