

TestIndex Benchmark

Embedded Database Systems for Android

Developers can now choose between database models, APIs, index types, and more, in small footprint DBMSs for mobile devices.

Which choices will result in the fastest, most efficient mobile software?

Overview

Mobile devices must store and manipulate growing volumes of complex data. Examples include location-based services (LBS) that rely on GIS data, MP3 player software that indexes songs and playlists, and field sales and service applications that store enterprise data.

Incorporating an off-the-shelf embedded database for these tasks reduces coding and QA, and improves reliability and performance. Given mobile devices' limited processing resources, though, the chosen database must be highly efficient. Optimal database performance results in a more satisfied end-user (who faces less waiting for information) and preserves memory and CPU cycles for other device functions.

TestIndex Benchmark Measures Efficiency

McObject's TestIndex Benchmark for the Android mobile phone environment compares embedded databases' performance on the following "generic" database tasks:

- Inserting 10,000 simple records (with string and integer keys, using two B-tree indexes) in the database. This result is reported as Insert time.
- Searching, via the indexes, for these records (10,000 searches for random values in each index, so the search is performed 20,000 times). This result is reported as Search time.
- Performing two traverses (sequential scans) through all records using both indexes (so records are sorted by integer key and by string key). This result is reported as Scan time.
- Locating and deleting all 10,000 records.

Perst vs. SQLite

The TestIndex Benchmark was used to compare McObject's open source, object-oriented Perst embedded database to SQLite, an open source, relational SQL embedded database. Tests were run in both the Android emulator and on the Android-based T-Mobile G1 smartphone. In Android's Java environment, TestIndex interacted with the all-Java Perst via the database's standard application programming interface (API), and with SQLite (a C application) using Android's Java API for the database.

In order to "compare apples to apples," identical table and index definitions were used. For SQLite, these took the following form:

```
create table TestIndex (i integer(8), s text);
create index StrIndex on TestIndex (s);
create index IntIndex on TestIndex (i);
```

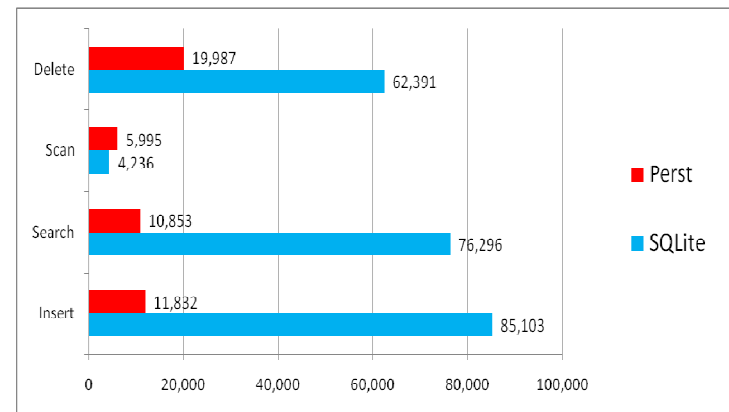
Perst's comparable setup was:

```
/*
 * Record used in this example contains just
 * two fields (columns)
 */
class Record extends Persistent {
    String strKey;
    long   intKey;
};

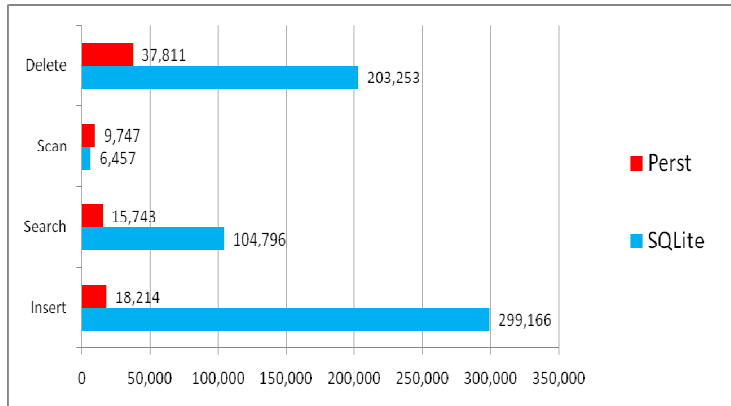
/*
 * This is the root object for the storage
 * containing indexes for both keys
 */
class Indices extends Persistent {
    Index<Record> strIndex;
    Index<Record> intIndex;
}
```

Benchmark Test Results

Comparing Perst to SQLite in Android's emulator returned the following results, measured in milliseconds:



In the T-Mobile G1 phone, TestIndex produced these results, in milliseconds:



Discussion

What accounts for the performance disparity? For SQLite insert and delete operations, one obvious gating factor is the lack of explicit transaction support in its Android API. Each update must be performed as a separate transaction, in autocommit mode, resulting in significant transaction processing overhead.

This also compromises the ability to maintain the logical consistency of the database contents (i.e. to define a database unit-of-work that consists of updates to two or more table rows in the database). For example, the classic bank account transfer could not be implemented within a single transaction that all succeeds or fails together:

```
update accounts set bal = bal-100 where acct_nbr = 123;
```

```
update accounts set bal = bal+100 where acct_nbr = 456;
```

Search time for SQLite is more than 6 times slower and is explained by overhead added by using Android's Java interface to access the native C language database, and the overhead of parsing, optimizing and executing the interpreted SQL. In contrast, Perst's interface works directly with database objects – no interpretation of an intermediate language is needed.

SQLite's advantage in scan operations probably stems from the test's simple tabular data layout. As a relational database, SQLite organizes data in rows, and these rows are physically close to one another in storage, like rows of a spreadsheet. This proximity lends an edge in sequentially fetching the rows. In contrast, in Perst, everything is an object, including index pages, and objects are interleaved in the storage.

The following factors also affect these databases' performance and their "fit" with mobile applications, and are worth considering:

Object-oriented vs. relational database system. Choice of database model is often viewed as hinging on the user's programming philosophy or style. But developers targeting resource-constrained devices should bear in mind the run-time efficiency gained by pairing an object-oriented database with an object-oriented language such as Java. In Java, developers work with Java objects; an object-oriented database stores these *as Java objects*, eliminating the translation required for storage in a relational or object-relational database.

This boosts run-time performance. It also permits the developer to stay within the O-O paradigm, whereas use of a relational database with an SQL interface requires shifting back and forth between O-O and a set-based, declarative query language.

Available database indexes. The B-tree used in TestIndex is by far the best-known database index, and is supported in essentially all database systems. It is efficient for general purpose database operations such as exact match, prefix and range searches. However, many specialized application types benefit from using "non-vanilla" indexes, such as the following:

<u>Application Type</u>	<u>Specialized Index</u>
GIS/Mapping	R-tree
Telecom/IP Routing	Patricia trie
In-memory application	T-tree

Conclusion

Mobile devices such as smartphones manage increasingly large data sets, to meet widely varying application needs. Enterprise and business-oriented database systems typically outstrip such devices' CPU and memory resources. However, several small-footprint embedded DBMSs have emerged, providing developers with the luxury of choosing between data models, APIs, index types and other database features for applications that will run on Android, BlackBerry, Windows Mobile, Symbian and other mobile devices. With this choice comes the ability to optimize data management based on application function and performance needs.

The TestIndex benchmark presented above provides a starting point for gauging embedded databases' efficiency. Full TestIndex source code is available for free download at http://www.mcobject.com/perst_eval.