Firebird for the database expert: episode 1 - Indexes

Youtube Tutorial Available

By Ann Harrison

Firebird differs in significant ways from other relational database management systems. Understanding the differences will allow you to create better-performing Firebird applications.

Audience: Experienced database application developers.

Moving to Firebird can be disconcerting for developers who have worked with other relational database management systems. In theory, relational databases separate the logical design of an application from the physical storage of the data, allowing developers to focus on what data they want their applications to access, rather how the data should be retrieved. In practice, the mechanics of each database management system make some styles of access much faster than others.

Developers learn to use methods that work with the database management systems they know. Developers who are familiar with Oracle or Microsoft SQL Server find that Firebird indexes, concurrency model, and failure recovery behave differently from the databases they know. Understanding and working with those differences will make your move to Firebird less stressful and more successful. This paper focuses on the unusual characteristics of Firebird indexes.

Index types

Firebird supports only one index type: a b-tree variant. Indexes can be unique or allow duplicates; they can be single key or compound key, ascending or descending.

Record location

Many databases cluster records on the primary key index, either directly storing the data in the index or using the key to group records. In a well-balanced system clustering on primary keys makes primary key lookup very efficient. If the full record is stored in the index, the data level becomes very wide, making the whole index deep and more expensive to traverse than a shallower, denser index. Record clustering can result in sparse storage or overflows depending on the design specifications and data distribution.

Firebird stores records on data pages, using the most accessible page with sufficient space. Indexes are stored on index pages and contain a record locater in the leaf node. Access costs of primary and secondary indexes. When data is clustered on the primary key, access by primary key is very quick. Access through secondary indexes is slower, especially when the secondary key index uses the primary key as the record locater. Then a secondary index lookup turns into two index lookups. In Firebird, the cost of primary and secondary index lookups is identical.

Index access strategy

Most database systems read an index node, retrieve the data - this technique also leads to bouncing between index pages and data, which can be resolved by proper placement control, assuming that the DBA has the time and skill to do so. For non-clustered indexes this technique also results in rereading data pages.

Firebird harvests the record locaters for qualifying records from the index, builds a bitmap of record locaters, and then reads the records in physical storage order.

Index optimization

Because their access strategy binds index access and record access tightly, most database optimizers must choose one index per table as the path to data. Firebird can use several indexes on a table by 'AND'ing and 'OR'ing the bitmaps it creates from the index before accessing any data.

If you have a table where several different fields are used to restrict the data retrieved from a query, most databases require that you define a single index that includes all the fields. For example, if you are looking for a movie that was released in 1964, directed by Stanley Kubrick, and distributed by Columbia you would need an index on Year, Director, and Distributor. If you ever wanted to find all pictures distributed by Stanley Kubrick, you would also need an index on Director alone etc. With Firebird, you would define one index on Director, one on Distributor, and one on ReleaseDate and they would be used in various combinations.

Long duplicate chains

Some databases (Firebird 2 for one) are better than others (Firebird 1.x for one) at removing data from long (>10000) duplicate chains in indexes. If you need an index on a field with low selectivity for a Firebird 1.x database, create a compound key with the field you want to index first and a more selective field second. For example, if you have an index on DatePaid in the table Bills, and every record is stored with that value null when the bill is sent, then modified when the bill is paid, you should create a two-part index on DatePaid, AccountNumber, instead of a single key index on DatePaid.

Indexes in lieu of data

Non-versioning databases resolve some queries (counts for example) by reading the index without actually reading the record data. Indexes in Firebird (like Postgres and other natively versioning databases) contain entries that are not yet visible to other transactions and entries that are no longer relevant to some active transactions. The only way to know whether an index entry represents data visible to a particular transaction is to read the record itself.

The topic of record versions deserves a long discussion. Briefly, when Firebird stores a new record, it tags the record with the identifier of the transaction that created it. When it modifies a record, it creates a new version of the record, tagged with the identifier of the transaction that made the modification. That record points back to the previous version. Until the transaction that created the

new version commits, all other transactions will continue to read the old version of the record.

In the previous example, when a transaction modifies the indexed field DatePaid, Firebird creates a new version of the record containing the new data and the identifier of the transaction that made the change. The index on that field then has two entries for that record, one for the original NULL value and one for the new DatePaid.

The index does not have enough information to determine which entry should be counted in responding to a query like "select count (*) from Bills where DatePaid is not null".

Index key length

In Firebird Version 1.x, the total length of an index key must be less than 252 bytes. Compound key indexes and indexes with non-binary collation sequences are more restrictive for reasons described in the section on key compression. Firebird 2 allows keys up to 1/4 of the page size, or a maximum of 4Kb.

Index key representation

Firebird converts all index keys into a format that can be compared byte-wise. With the exception of 64bit integer fields, all numeric and date fields are stored as double precision integer keys, and the double precision number is manipulated to compare byte by byte. When performing an indexed lookup, Firebird converts the input value to the same format as the stored key. What this means to the developer is that there is no inherent speed difference between indexes on strings, numbers, and dates. All keys are compared byte-wise, regardless of the rules for their original data type.

mIndex key compression

Firebird index keys are always stored with prefix and suffix compression. Suffix compression removes trailing blanks from string fields and trailing zeros from numeric fields. Remember that most numeric values are stored as double precision and so trailing zeros are not significant. Suffix compression is done for each key field in a compound key without losing key field boundaries. After removing the trailing blanks or zeros, the index compression code pads field to a multiple of four bytes, and inserts marker bytes every four bytes to indicate the position of the field in the key.

Consider the case of a three field key with these sets of values:

```
"abc","def","ghi"
"abcdefghi","",""
```

Simply eliminating trailing blanks would make the two sets of values equal. Instead, Firebird turns the first set of key values into "abc 1def 2ghi 3" and the second into "abcd1efgh1i 1 2 3".

Firebird version 1.x compresses the prefix of index keys as they are stored on pages in the index. It stores the first key on a page without prefix compression. Subsequent keys are stored after replacing the leading bytes that match the leading bytes of the previous key with a single byte that contains the number of bytes that were skipped. The two keys above would be stored like this:

```
"0abc 1def 2ghi 3" "3d1efgh1i 1 2 3"
```

An index entry that exactly matches the previous entry is stored as a single byte that contains its length. Firebird 2 also performs prefix compression, but uses more dense representation. The combination of compression techniques eliminates some of the rules about constructing keys. Suffix compression occurs on all segments of a key, so long varchar fields should be placed in their logical spot in a compound key, not forced to the end. On the other hand, if part of a compound key has a large number of duplicate values, it should be at the front of a compound key to take advantage of prefix compression.

This paper was written by Ann Harrison in June 2005, and is copyright Ms. Harrison and IBPhoenix.

From:
http://ibexpert.com/docu/ - **IBExpert**

Permanent link:
http://ibexpert.com/docu/doku.php?id=01-documentation:01-05-database-technology:database-technology-articles:firebird-interbase-ods:firebird-for-the-database-expert-indexes

Last update: **2023/06/16 12:24**