



Impact of Key Locality on Performance

A CopperEye Technical White Paper

January 2004

Introduction

CopperEye indexing is a general-purpose data indexing technology that significantly out-performs conventional indexing techniques. This paper discusses the reasons behind the superior performance of this technology over alternative methods.

Background

Indexes are used in database and data storage/retrieval applications to enable fast access to stored data. Without indexes, applications are forced to scan entire data sets to find the required data. But while indexes speed up data retrieval, they incur a significant overhead during data set updates and a compromise must often be struck between data retrieval performance and data loading performance.

Indexes can be categorised according to the data types they handle. For example, an index may be used for accessing general-purpose scalar data, such as text, number and date information. Alternatively, an index may be designed for handling structured data formats such as spatial or media data. However, the vast majority of data held in commercial database systems is of a scalar type and scalar indexes are predominant across all industries.

Indexes can also be categorised according to the resources they are optimised for – memory or disk. For example, an index may reside entirely in memory and/or may access entirely memory resident data. Such an index aims to minimise its memory footprint and CPU usage. Whereas a disk-resident index focuses on reducing its disk access, since disks offer a slow medium for data access. Memory based indexes restrict both database growth and tolerance of system failure. Therefore, most commercial databases use disk resident indexes to provide sufficient database size and security.

The CopperEye technology is a general-purpose scalar disk-optimised index and is suited to the majority of data stored in commercial systems. For both brevity and clarity, specialist multi-dimensional data indexes and memory-optimised indexes will not be considered any further in this paper.

Throughout this paper, reference is often made to relational databases because the reader is likely to be familiar with them. However, the CopperEye technology is not specific to relational databases and can apply to any data storage paradigm, such as object-oriented databases, XML databases or flat file systems.

Industry Trends

Few radical innovations in disk-optimised scalar indexing have occurred over recent decades. Commercial database vendors have concentrated on improving performance through hardware scalability and operational parallelism, which provide performance at a hardware cost; but even then, not all database applications are open to divide-and-conquer approaches.

The technology innovations that do address scalar data tend to optimise query performance at great expense to loading performance and concurrency.

Meanwhile, commercial database sizes are burgeoning and enterprises expect to exploit their databases more effectively. This exerts pressure on database performance and particularly the index structures used to navigate through the sea of data.

The CopperEye technology alleviates exactly these performance issues.

Key Localisation

Key localisation is the measure of how and where key instances are placed into a storage structure (such as a table or index) in relation to their key values. Key localisation can be categorised by two dimensions – precision and clustering. Precision dictates the probability of knowing exactly where a key instance is located in the structure given the key value. The greater the precision, the more precisely the location is known, and the more efficiently a search can return an instance of a specific search key. Clustering dictates the probability of finding multiple instances of similar keys within close proximity of each other. The greater the clustering, the less scanning is required and the more efficiently multiple instances can be returned for a range of keys.

A heap-organised table, wherein new rows are simply appended at the end of the structure, has the least precise key localisation and no clustering because the key value has no influence on its row locality. In contrast, a hash-organised table, where rows are placed at specific addresses derived from a hash algorithm applied to the associated key, has high precision but no clustering – instances of similar (but different) keys are unlikely to be in close proximity to each other. Whereas a B-tree, with a hierarchy of blocks grouping keys together in key order, exhibits both high precision and clustering – instances of similar key values are likely to be found in the same index block.

These different mixtures of key localisation clearly afford different search efficiency. The heaped table requires a full table scan to either locate an instance of a single key or to locate multiple instances for a range of keys; while the hashed table can efficiently locate an instance of a specific key, but still requires a full table scan to locate a range of keys; whereas a B-tree can perform both searches efficiently.

Performance

The most costly operations for a disk-resident storage structure involve write and read access to and from the disk; where clearly the fewer disk accesses yield better performance. The pattern of disk access is largely determined by the precision of key localisation within the structure; less precision yields easier structure maintenance but affords less efficient key searching and visa versa.

The structural features of common indexes, such as the B-tree or hashed table enforce highly precise localisation to achieve the best possible search efficiency. However, the extreme precision of these structures makes each of them expensive to maintain. Contrast the maintenance cost of a heaped table with a hashed table; multiple rows with randomly distributed keys can potentially be appended to a heaped table with a single disk write at the table high watermark (Fig. 1); whereas a hashed organised table can demand up to two disk operations per row inserted (one disk read to fetch the target table block encompassing the

row address followed by a disk write to persist the updated block) as shown in Fig. 2. These are dramatically different maintenance cost profiles and not surprisingly they are inversely proportional to their search efficiencies. The B-tree similarly exhibits high precision localisation and can require two disk operations (or more) for each row inserted (Fig. 3).

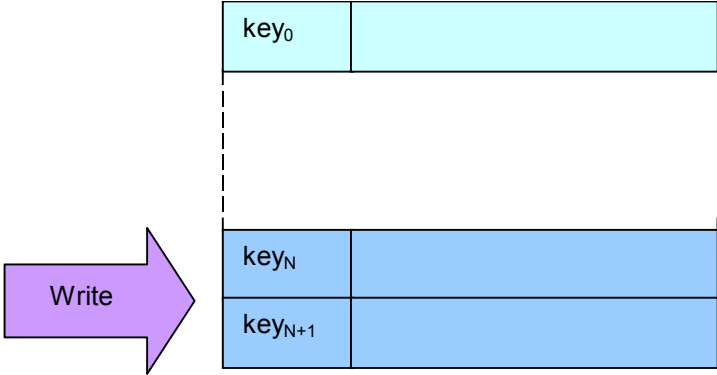


Fig 1. Inserting 2 rows into a heap organised table

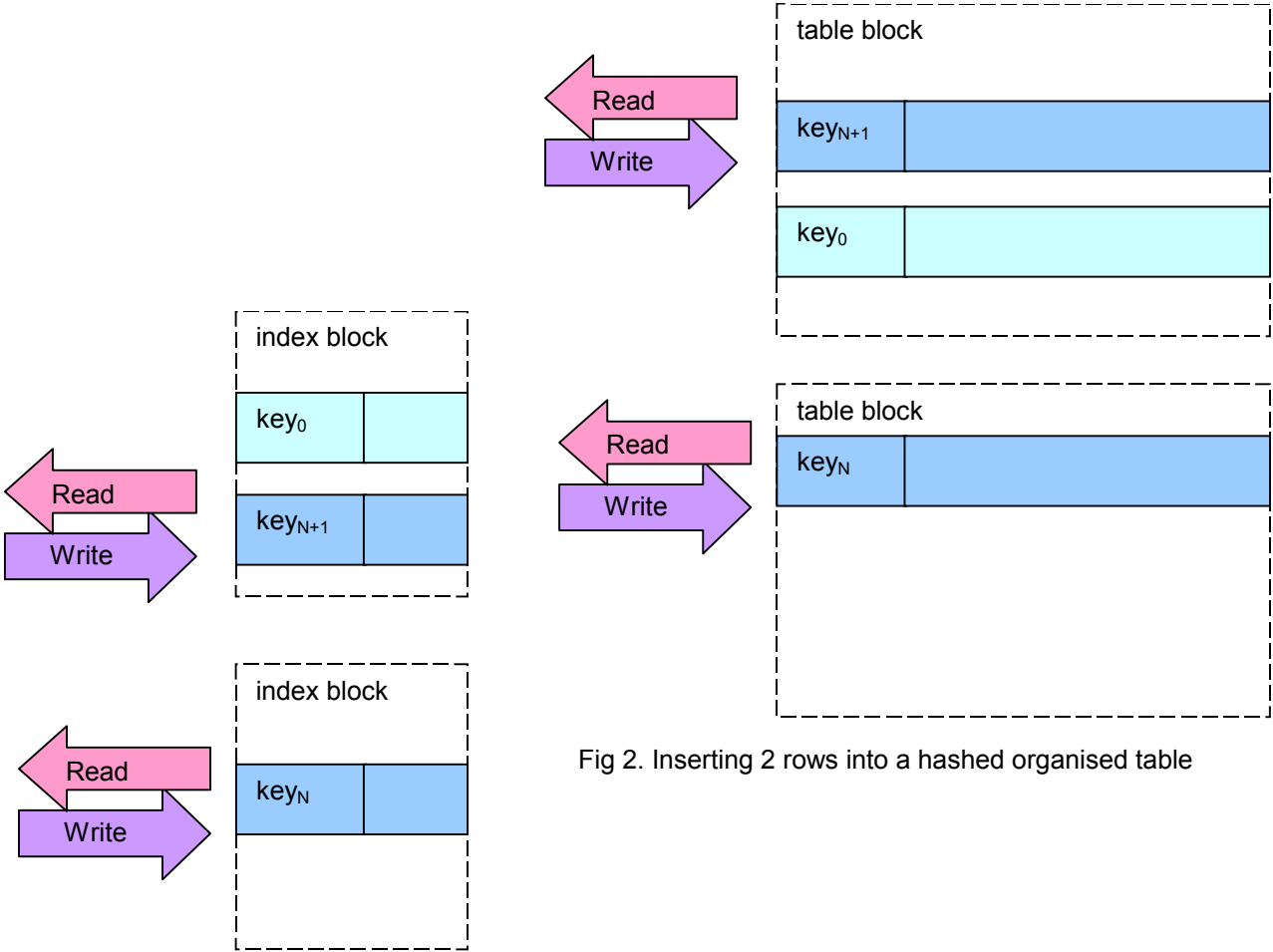


Fig 2. Inserting 2 rows into a hashed organised table

Fig 3. Inserting 2 rows into a B-tree

Perversely, the most common arrangement of a heaped table with a B-tree index on top, gives the worst possible performance all round. The B-tree stifles row inserts, deletes and key updates; while the heaped table dampens range query speed by forcing a disk read for each row returned (Fig. 4). This is partially overcome by B-tree organised tables (whereby the entire table rows are stored within the B-tree structure itself) but only insofar as queries are serviced quickly; meanwhile, the overhead of row inserts, deletes and key updates remains high – indeed it actually worsens because fewer keys can be accommodated in each index block, requiring more frequent internal reorganisation activity.

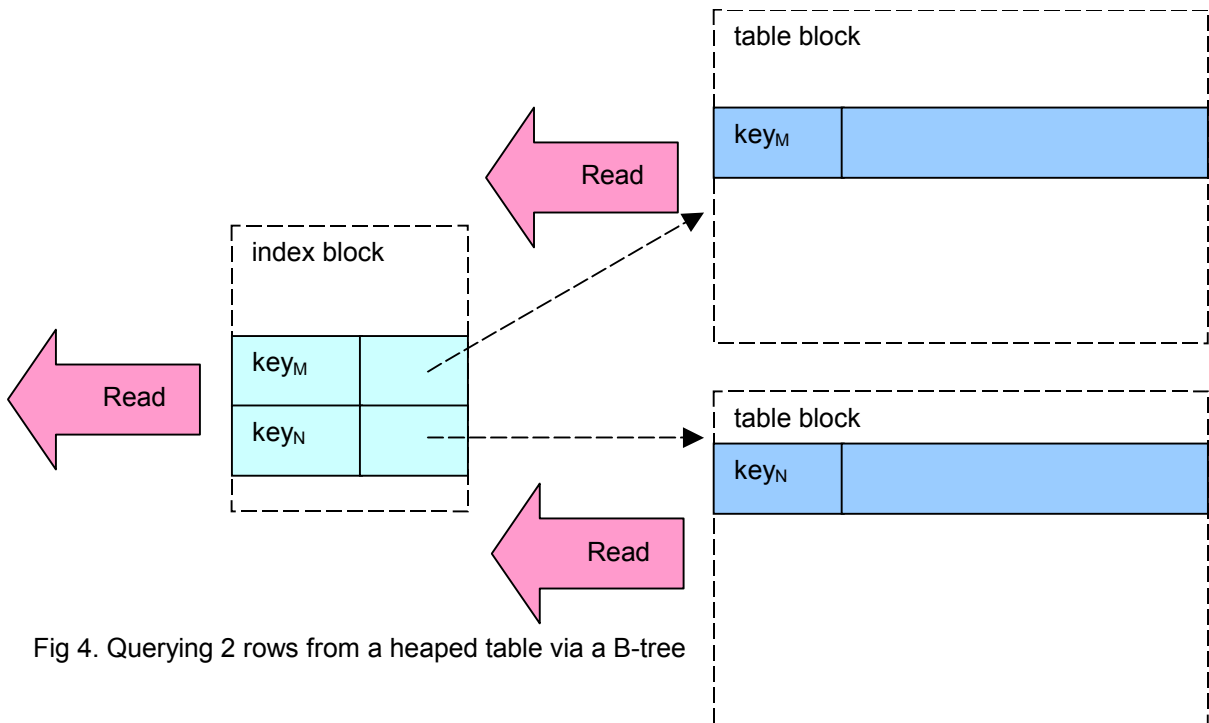


Fig 4. Querying 2 rows from a heaped table via a B-tree

Moreover, the B-tree organised table can only be built around one key (typically the primary key), requiring secondary indexes to be built on top to support access by secondary keys. While this arrangement affords fast primary key searches, it degrades access by secondary key still further because of the increased complexity involved in navigating the B-tree organised table over the heap organised table.

Flexibility

The localisation precision is the prime determinant of the structure maintenance overhead, whilst clustering mainly affects the efficiency of key range searches. As we have already seen, the precision of B-trees and hashed tables are similar and exhibit similar maintenance costs despite having very different clustering characteristics. Therefore, high clustering is always desirable if it does not enforce highly precise localisation too. Clustering only influences maintenance

performance when pre-ordered data is being inserted into the structure. In these circumstances, high clustering is desirable and yields good insert performance; hence when building B-trees on pre-populated tables, the keys are usually sorted first.

The CopperEye index deploys a structure that can vary the localisation precision at will and maintain a commensurate key clustering. The reduced precision allows unrelated key instances to share disk write operations where scopes of locality overlap (Fig. 5). When built on top of a heap-organised table, the localisation precision of the index can be relaxed to avoid excessive maintenance overhead for row inserts, deletes and key updates while providing query performance suited to the underlying table. Moreover, localisation precision can be adjusted to reflect the access profile of the applications using the index. For example, an application that is predominantly insert-intensive with occasional queries (such as a customer transaction fraud monitoring application) can use indexes with a relaxed precision to ensure that the customer transactions are loaded and indexed as quickly as possible. This is in contrast to a B-tree or hashed table, where the fixed precision dictates that the only route to better throughput is through hardware expansion and/or a parallel architecture.

A CopperEye index can also store the table rows internally, providing an alternative to the B-tree organised table. This arrangement can provide much faster range query access than a conventional B-tree and heaped table combination, while still affording very fast row insert, delete and key update performance. Naturally, additional CopperEye indexes can be built on top of a CopperEye table to support secondary key access.

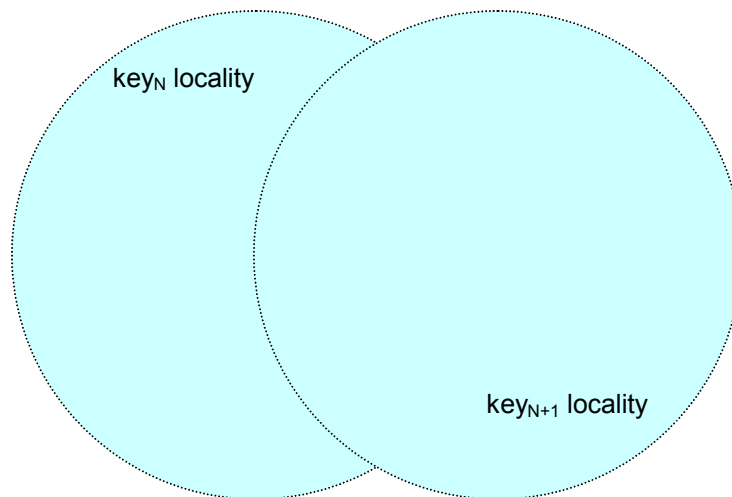


Fig 5. Relaxed precision affords shared disk access

Implications

Any index that proves burdensome to maintain will be used sparingly and limit the queries it can effectively service. Database design often has to compromise query performance against overall performance – so few database schemas can

implement a comprehensive arrangement of indexes to satisfy virtually any query. An important implication of the more efficient index maintenance possible with CopperEye technology is that it affords index proliferation, which in turn promotes query flexibility.

Furthermore, poor maintenance performance often forces undesirable architectural decisions to be made such as query lockout during load activities and out-of-hours batch loading windows instead of continuous incremental streaming. More efficient index maintenance afforded by CopperEye technology can yield a simpler architecture.

Further Information

For further information, please contact marketing@coppereye.com.

UK Corporate Headquarters

CopperEye Ltd, Suite 47, Aztec Centre, Almondsbury, Bristol BS32 4TD
t +44 (0) 1454 203610 f +44 (0) 1454 203330

US Headquarters

CopperEye, 101 Federal Street, Suite 1900, Boston, MA 02110, USA
t +1-617-342-7173 f +1-617-342-7080

e contact@coppereye.com
w www.coppereye.com

©2010 CopperEye, Ltd. All rights reserved. CopperEye and the CopperEye logo are trademarks of CopperEye, Ltd.