Profile of CopperEye Indexing Technology

A CopperEye Technical White Paper

Introduction

CopperEye's has developed a new general-purpose data indexing technology that out-performs conventional indexing techniques. This paper describes the new indexing technology and how it can be exploited.

CopperEye indexing has many aspects that appeal to application developers and database administrators alike; and this new technology overcomes many of the drawbacks experienced with incumbent index technology.

This document discusses the general characteristics of indexes and how this new technology compares with the currently dominant index structures.

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 technology 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, this new 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.

Dominant Indexes

The current dominant general-purpose scalar indexes used in commercial disk based database systems today are

- B-Tree indexes
- Hashed indexes
- Bit map indexes

The characteristics of these index structures are compared against CopperEye indexing in this paper to position this new indexing against the incumbent technology. It is assumed throughout this document, that the reader is reasonably familiar with the structure of the above index types.

Each of the standard index structures has a number of variants that are used throughout industry. For example, the B-Tree often surfaces as a B+ Tree or B* Tree variant. However, from a general functionality and overall performance standpoint, there is little to distinguish the variants and this paper will focus its attention on the fundamental structure of each index type.

Each of the dominant index structures provides varying levels of functionality and performance. A database vendor will typically provide one or more of these as standard index types, which can be applied to data stores within the database. An application developer must choose a suitable index according to an application's functional, performance and operational requirements. Often, a developer is forced into a compromise decision because of conflicts between the application requirements and index performance constraints.

CopperEye indexing is a new alternative index structure that out-performs the current dominant structures and provides a powerful addition to the armoury of indexing techniques available to developers and database administrators.

Index Framework

An index provides a mapping from data keys to data addresses. The key labels the data and the address identifies its location. For example, a key may be a customer name and the address is the location of the customer row within a customer table. The index maps from one or more keys to zero, one or more addresses.

An index must support the following minimal interface in order to provide a useful mechanism for indexing scalar data:

- Insert a key and address pair into the index mapping.
- Delete a key and address pair from the index mapping.
- Retrieve a set of addresses from the index mapping that are associated with a key specification.

The insert or deletion of a key-address pair in an index structure is known as an index maintenance operation. Scanning or navigating an index structure to find key-address pairs is known as an index retrieval operation.

Within this common framework, indexes can vary enormously in their structure and access algorithms. These variations lead to differences in the efficiency of index maintenance and retrieval operations; and differences in how keys are specified for retrieval operations.

Index Characteristics

When comparing index structures, it is useful to consider the following characteristics

- Key Behaviour. How successfully the index handles a diversity of key data.
- Retrieval Flexibility. What retrieval restrictions are enforced by the index
- Performance Scalability and Concurrency. How the index performs as data volumes and usage grows.
- Disk Binding. How the index mitigates disk performance.
- Resource Footprint. How much memory and disk space is consumed by the index.
- Workload Adaptability. How the index can adapt to application requirements.

Each of the above is considered in detail below for each index structure in the following sections.

Key Behaviour

Keys vary, but not all index structures are suitable for all key types and distributions. For example, Bit Map indexes do not handle high cardinality keys well.

The aspects of key behaviour that influence index suitability include

- Type. The data type and how it is represented.
- Size. The maximum key size applicable.
- Cardinality. The proportion of distinct key values across the key population.
- Distribution. The clustering of key values across the key domain.

The aptitude of an index for certain key characteristics is fundamentally determined by its structure. For example, indexes that store the native key value internally are likely to suffer performance degradation with large key sizes, because of the direct impact on index size.

The table below presents a chart of index suitability against key behaviour.

Behaviour	CopperEye	B-Tree	Hash	Bit Map
Character Key	Yes	Yes	Yes	Yes [1]
Number Key	Yes	Yes	Yes	Yes [1]
Date Key	Yes	Yes	Yes	Yes [1]
Composite Key	Yes	Yes	No [2]	No [2]
Large Key Size [7]	Yes	No [8]	Yes	Yes
Low Cardinality	Yes [3]	Yes [3]	No [4]	Yes
High Cardinality [5]	Yes	Yes	Yes	No
Highly Clustered	Yes	Yes	No [6]	Yes

Notes:

[1] *A Bit Map is only suitable if the key domain is sufficiently small or can be surrogated to a smaller domain.*
[2] *Composite keys can only be handled by surrogating the composite key combination, which restricts the key combinations that can be retrieved.*
[3] *The index structure can adequately handle low cardinality keys, but may not be as efficient as a bit map index.*
[4] *A hash index will typically suffer collision problems with low cardinality key data.*
[5] *High cardinality is assumed to be over 2^{16} (64K) distinct potential key values.*
[6] *A hash index will typically suffer collision problems with highly clustered key data.*
[7] *A large key size is assumed to be over 32 bytes long.*
[8] *A B-Tree stores multiple native keys in a disk block. As the key length increases and the number of keys stored per block reduces, the overall performance of the index degrades.*

The table illustrates that B-Trees do not tolerate large keys; while hash indexes can suffer performance problems with low cardinality or clustered key distributions; and bit map indexes do not handle high cardinality keys well. However, CopperEye indexing provides good all-round suitability regardless of the key characteristics.

Retrieval Flexibility

The possible key specifications for retrieving data addresses from an index include:

- Exact Key Match. Only keys that exactly match the specified key are returned. For example, return all employees whose department is “SALES”.
- Range Match. All keys that fall within the specified range are returned. For example, return all employees whose salaries are between $20,000 and $50,000.
- Pattern Match. All keys that match the specified key wildcard are returned. For example, return all employees whose surnames are like SMITH or SMYTHE etc.

- Token Match. All keys that contain the specified tokens are returned. For example, return all employees whose skill description includes the words "C++" and "ORACLE".

An index will support one or more of these specifications, but typically not all.

The retrieval methods supported by an index are fundamentally determined by its internal structure. The table below presents a summary of the retrieval options available for each index.

Retrieval	CopperEye	B-Tree	Hash	Bit Map
Exact Match [1]	Yes	Yes	Yes	Yes
Range Match [2]	Yes	Yes	No	No [5]
Pattern Match [3]	Yes	No	No	No
Token Match [4]	Yes	No	No	No

Notes:

[1] *An SQL example of an exact match is*

```
SELECT…WHERE column=value.
```

[2] *An SQL example of a range match is*

```
SELECT…WHERE column BETWEEN min AND max.
```

[3] *An SQL example of a pattern match is*

```
SELECT…WHERE column LIKE wildcard.
```

[4] *An SQL example of a token match is*

```
SELECT…WHERE INSTR(column,' '||includedtoken||' ')>0
  AND INSTR(column,' '||excludedtoken||' ')=0
```

[5] *Bit Map indexes can potentially support range matching, but its low cardinality key restriction usually makes it pointless.*

It is clear that CopperEye indexing provides the greatest flexibility for key retrieval, whereas Hash and Bit Map are the most restrictive.

Performance Scalability and Concurrency

The size of an index structure typically grows with the volume of key data mapped and, as a result, the efficiency of the index is likely to degrade. If performance does degrade significantly, this should ideally happen gracefully, or at the very least, predictably. The features of an index structure that most affect performance are:

- Growth. The rate of structural growth against increasing key volumes. The less an index grows the less impact on its performance.
- Maintenance Scope. The extent of the structure affected by maintenance operations. The more extensive any change in index structure, the more impact there is on its maintenance overhead.
- Retrieval Scope. The extent of the structure scanned by retrieval operations. An index that scans a greater proportion of its structure will be more affected by index growth.

The scalability of an index is largely decided by how the index structure grows; how much of an index structure is affected by an index maintenance operation; and how much of an index structure must be scanned during a retrieval operation. Indexes that grow rapidly and require extensive structural re-organisation

typically scale badly for index maintenance operations. Indexes that grow rapidly and require exhaustive structure scanning typically scale badly for index retrieval operations.

Performance scalability determines the suitability of the index in large volume applications and the level of administration or re-engineering required as volumes continually grow.

The concurrency of an index is largely determined by how much of an index structure is affected by an index maintenance operation. Indexes that require extensive structural re-organisation during index maintenance operations typically exhibit poor concurrency. This arises where large portions of the index structure must be locked away from other index users for the duration of a maintenance operation.

In multi-user or multi-process environments, concurrency can be a prime factor in the overall throughput of the system.

The table below presents a summary of the performance characteristics for each index.

Performance Factors	**CopperEye**	**B-Tree**	**Hash**	**Bit Map**
Order of Growth [1]	N	N.log(N)	N	log(N)
Maintenance Scope	Local	Local	Local/ Global [2]	Local
Retrieval Scope	Local	Local	Local	Global
Good Scalability	Yes	Yes	No [3]	Yes
Good Concurrency	Yes	Yes	No [3]	No [4]

Notes:

[1] *The order of growth is the relative change in structure size compared to a change in key volume size (N). The calculations represent rough estimates of behaviour, rather than specific size calculations.*
[2] *A hash index has a maximum predetermined capacity which, when exceeded, requires an index rebuild and hence exhibits occasional global maintenance scope.*
[3] *A hash index exhibits good volume scalability and concurrency until its predetermined capacity is exceeded.*
[4] *Most bit map implementations exploit some form of data compression. This compression creates dependency between areas of the index and increases the scope of maintenance operations.*

CopperEye indexing provides a good order of growth with local scope for both maintenance and retrieval operations; and hence exhibits good scalability and concurrency.

Disk Binding

A disk resident index naturally incurs disk I/O costs during maintenance and retrieval operations. Given the high latency of disk operations, this tends to dictate the performance aspects of the index. A typical mechanical hard disk cannot support more than about 250 random I/O operations per second. Therefore, it is important to keep the number of disk read and write requests low for the maintenance or retrieval of any given key-address pair.

When an index size is sufficiently small to fit within available physical memory, the disk bound nature of the index is likely to be masked by cache effects. Moreover, an index with insufficient memory will incur unnecessary and excessive disk I/O traffic. Therefore, it is assumed here that each index operates within its nominal memory configuration.

The table below presents a summary of the disk I/Os incurred, averaged over each key-address pair inserted and queried from maintenance and retrieval operations respectively. It is assumed that keys are inserted and retrieved in random order.

Operation	CopperEye	B-Tree	Hash	Bit Map
Maintenance [3]	< 1 [11]	2+ [1]	2 [2]	< 1 [6]
Retrieval Min [4]	(A+CK)/B	(A+K)/B	0 [5]	C/B [8]
Retrieval Max [7]	(3log(N(A+CK))-14)/S [9][10][12]	1	0 [5]	NC/B [8]

Legend:

A = address size
B = index block size
C = compression factor (between 0 and 1)
K = key size (total across all composite segments)
N = number of key-address pairs
S = bias (typically between 3 and 8)

Notes:

[1] *To insert a key-address pair requires a read I/O to fetch the relevant leaf block and a write I/O to update it with the new pair (2 I/Os). Additional I/Os may be required to effect changes in upper levels of the tree.*
[2] *A hash index may either store addresses in independent hash clusters or may enforce hash clustering of the underlying data store. Either way, inserting a key-address pair will require a read I/O to fetch the relevant hash cluster and a write I/O to update it with the new pair (2 I/Os).*
[3] *A maintenance operation to insert a single key-address pair. The I/O figures relate purely to index overhead.*
[4] *A best-case retrieval operation that retrieves multiple key-address pairs to amortized the I/O cost. The I/O figures relate purely to index overhead.*
[5] *This assumes hash clustering of the underlying data store. If addresses are stored in independent hash clusters then at least 1 additional read I/O will be incurred.*
[6] *Index I/O cost is usually amortized across multiple key-address pairs because of the compact nature of the index and especially where inserts are concentrated in a data store locality, such as when appending consecutive entries.*
[7] *A worst-case retrieval operation that retrieves a single key-address pair. The I/O figures relate purely to index overhead. Typically this figure is between 1 and 3.*
[8] *The effective compression factor C depends on the key encoding method and the sequence of keys involved.*
[9] *The compression factor C depends on structure of the key and the retrieval methods required. It can be as low as 0.004 (i.e. 0.4% of the original key space).*
[10] *The bias S is a dynamic factor that can be increased to favour retrieval efficiency over maintenance efficiency.*
[11] *Typically less than 0.01.*
[12] *Typically this figure is between 1 and 3.*

Thus CopperEye indexing is significantly less disk-I/O bound during index maintenance operations than either its B-Tree or Hash counterparts. CopperEye

index retrievals are also potentially less disk-bound than B-Trees for queries that return moderate or large numbers of key-address pairs.

Resource Footprint

A disk resident index occupies both disk space and memory. The following table gives approximate estimates for the disk and memory occupancy expected with each index type.

Resource	CopperEye	B-Tree	Hash	Bit Map
Memory Footprint	$N(A+CK)/ 2^{11-S}$ (5)(6)	$N(K+A)^2 /F^2B$ (1) (3)	0 (2)	B
Disk Footprint	2N(A+CK) (5)	N(K+A)/F (3)	NK/(1/F-1) (2)(3)	NC/B (4)

Legend:

A = address size
B = index block size
C = compression factor (between 0 and 1)
F = block fill factor (between 0 and 1)
K = key size (total across all composite segments)
N = number of key-address pairs
S = bias (typically between 3 and 8)

Notes:

(1) *Assumes that all blocks other than leaf blocks are in memory. However, this calculation only includes the penultimate level and assumes that higher levels are insignificant. Although the upper levels become more significant with large key sizes, this is likely to be compensated by key prefix compression, where used.*
(2) *Assumes hash clustering of the underlying data store.*
(3) *The block fill factor F is typically in the region of 0.75.*
(4) *The effective compression factor C depends on the key encoding method and the sequence of keys involved.*
(5) *The compression factor C depends on structure of the key and the retrieval methods required. It can be as low as 0.004 (i.e. 0.4% of the original key space).*
(6) *The bias S is a dynamic factor that can be increased to favour retrieval efficiency over maintenance efficiency.*

CopperEye indexing occupies the middle ground in resource efficiency, requiring more resources than Hash Indexes or Bit Maps, but typically less than a B-Tree. In general, a CopperEye index will consume less resources relative to a B-Tree when

- Key sizes are greater than 8 bytes long
- Retrieval flexibility is sacrificed
- Maintenance efficiency is favoured over retrieval efficiency

Workload Adaptability

Software products and applications vary in their functional and performance demands of an index. For example, a fraud application will typically need to continuously load large volumes of transactional data; but will only need to query transactions in response to infrequent suspicious activity.

Standard indexes offer little or no adaptability for workload requirements; whereas the structure of CopperEye indexing allows the following requirements to be controlled on an individual index basis:

- Retrieval flexibility can be balanced against resource and operational efficiency. It is possible to reduce query flexibility to achieve minimal resource usage and maximize operational performance.
- Maintenance efficiency can be balanced against retrieval efficiency. It is possible to increase maintenance performance at the expense of query performance or vice versa.
- Operational efficiency can be balanced against resource efficiency. It is possible to increase operational performance at the expense of resource efficiency or vice versa.

Summary

CopperEye has a new general-purpose indexing technology with an excellent performance profile. It allows application developers and database administrators to

- Reduce memory usage
- Reduce disk usage
- Proliferate indexes for better query performance with minimal loading performance impact
- Implement efficient composite key indexes
- Tune indexes to meet the workload

These aspects promote overall faster application performance.

Intellectual Property

CopperEye technology is a proprietary indexing technology with the Intellectual Property Rights wholly owned by CopperEye Ltd and protected by multiple worldwide patents.

Further Information

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