



# **Using the CopperEye SDK**

---

*A CopperEye Technical White Paper*

*May 2004*

## Introduction

CopperEye indexing is a general-purpose data indexing technology that significantly out-performs conventional indexing techniques. This paper discusses how the CopperEye SDK can be used to implement this technology to achieve enormous performance gains in a software application.

## 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. Moreover, CopperEye technology is not specific to any particular data storage paradigm and can be used effectively regardless of the storage model used.

## 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.

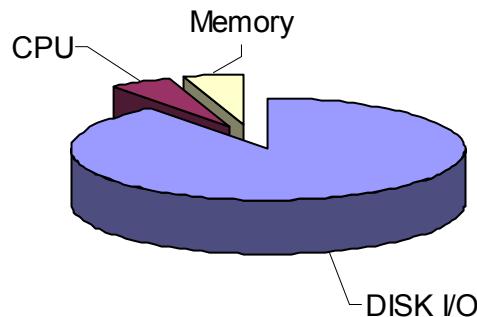
The technology innovations that do address scalar data tend to optimise query performance at great expense to loading performance and concurrency.

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

The CopperEye technology alleviates exactly these performance issues.

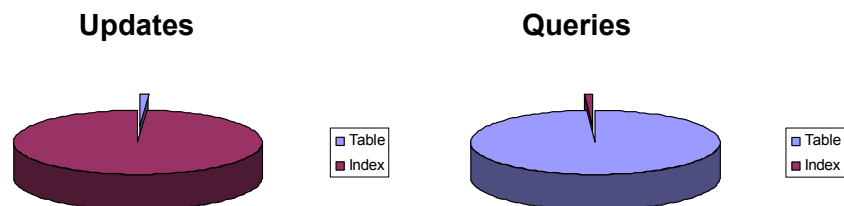
## Indexing Issues

Indexes are needed to locate data in an efficient manner. Without any indexing mechanism, a data store must be scanned across its entirety to find a particular piece of data and this is not viable for any sizable data store, which typically resides on disk. Disk I/O is slow and scanning a data store sequentially can be a time consuming exercise. Indexes allow a search to precisely pin-point the location of specific data placed within a data store and avoid the expensive sequential scans otherwise required to reach it.



**Fig 1. The Overhead of Disk I/O**

The greatest overhead of an index typically results from any disk I/O it incurs. While indexes are generally able to perform searches efficiently, incurring little overhead within the overall search process, they can be expensive to update and become disk bound as volumes grow – creating a performance ceiling for the rate at which data can change. Highly transactional systems commonly use minimal indexing to achieve a reasonable throughput; even data warehouses typically drop indexes and recreate them after the data load because the cost of updating the indexes incrementally as the data changes is far too great.



**Fig 2. The relative disk I/O cost of indexing for updates and queries.**

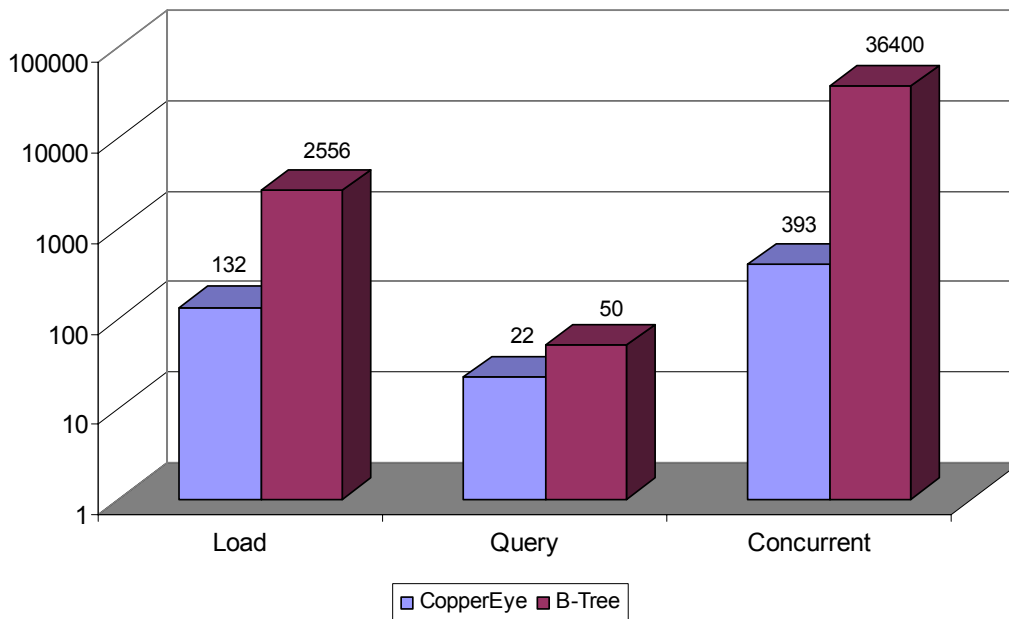
Multiple indexes may need to be built on top of a data store to help navigation into the data store from different search dimensions – the more flexible the search requirements, the more indexes are likely to be required. More indexes mean a greater overhead for updates and a reduced rate at which data can change – thus indexes can force a difficult compromise

between query flexibility and data acquisition. Indeed indexes create a significant hurdle to a mixed workload paradigm where powerful analytics are combined with rapid data acquisition.

## The Index Solution

The CopperEye SDK delivers indexing which is efficient for queries *and updates* – enabling indexes to be proliferated without hitting the performance wall traditionally associated with such an approach. A CopperEye index can be updated at up to a 100 times faster than a comparable B-tree with equivalent hardware resources, while still delivering excellent query performance; allowing many more indexes to be built and maintained when compared to conventional indexing options. The CopperEye SDK delivers the mixed workload paradigm by allowing rapidly changing data to be comprehensively indexed as the changes occur without recourse to excessive hardware architectures. More indexes yield greater query flexibility and speed while the rapid index updates ensure that query results are up to date and relevant.

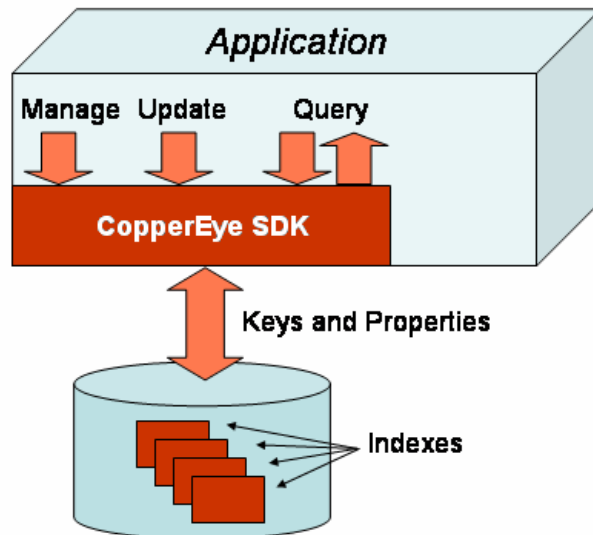
The CopperEye SDK ensures predictable and scalable performance on indexes that can contain many millions or billions of index entries and provides an environment within which rates of many thousands of transactions per second are readily achievable concurrently with flexible and powerful queries.



**Fig 3. Example of performance improvements actually achieved by replacing a B-Tree within an application. (Time taken for loading, queries and mixed concurrent workloads)**

## Implementation

The CopperEye SDK provides a comprehensive set of APIs available in C, C++ and Java that allow applications to embed CopperEye indexing directly within the application. The generic indexing framework offered by the SDK allows other general purpose indexing methods (such as B-tree and hashed indexes) to be directly replaced where they are currently used or indexing to be introduced where it is currently missing.



**Fig 4. Embedded SDK**

An index provides a framework for storing and retrieving key and property pairs whereby key and property pairs can be inserted and/or deleted from the index and subsequently retrieved from the index by matching key criteria. The key is simply a label used to identify the associated property while the property carries a payload that can contain application data, the address of the application data or a mixture of both. For example, an index may contain keys of customer names while the associated properties contain the file offsets to locate the associated customers stored in a separate file. A search for a particular customer can use the index to locate a customer name and use the file offset in the associated property to look up the customer data. Both keys and properties can be segmented into multiple fields allowing both to contain a concatenation of application data – thus a key may contain both a customer given name and family name, for example. Multiple keys can even be stored in a single index allowing multiple key searches for a common entity – thus a single index can be searched by customer name and location as separate keys.

Queries retrieve key and property pairs from indexes through one or more combined predicates that reflect the search criteria. Each predicate can look for exact, range or wildcard key criteria; both across the entire key and/or for partial sections of the key. The SDK also allows query results to be sorted by all or part of a key, or all or part of a property and provides the functionality to join the results from multiple queries together in a combined result with common key values. Queries can also contain a temporal element to return

query results from previous points in time to allow historic analysis and comparisons.

Multiple application threads can use the same index concurrently, with threads simultaneously inserting, deleting and retrieving data from an index. The SDK provides full transactional control with transactional isolation between threads, allowing multiple insert and/or delete operations to be committed or rolled back as an atomic operation. Operations performed by one thread are not visible to any other threads until the transaction enclosing the operations is atomically committed.

The SDK provides comprehensive functionality for managing the lifecycle of indexes, allowing them to be dynamically created, dropped, truncated and partitioned. Partitioning can be effected both physically across separate index files and logically within the same index file allowing data placement and lifecycle to be managed according to application requirements. Moreover, partitions can be dynamically created and dropped to support rolling data management.

Backup and recovery of indexes are supported through optional index mirrors, transaction control and index undo - to guard against media failure, process failure and application failure respectively. Index mirrors can provide a complete up-to-date copy of an index on a separate disk allowing an index to be rapidly retrieved in the event of media loss. Transactions provide the framework of atomicity to ensure that related operations are consistent and can be tied to the application state. Index undo operations allow indexes to be undone to a previously committed historic transaction to ensure the index remains consistent with the application even if the application is reversed to a previous application state. The index undo facility also avoids the need for a slow and expensive two-phase commit architecture in a distributed data environment.

## Platforms

CopperEye products are implemented in C/C++ and Java, and are available in all tier-one UNIX environments, namely

- Sun/Solaris
- IBM/AIX
- Compaq/True64
- HP/HP-UX
- Linux

## 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](mailto: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](mailto:contact@coppereye.com)  
w [www.coppereye.com](http://www.coppereye.com)