CSCI317 Database Performance Tuning

Introduction to Indexing

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Introduction to Indexing

Outline

Index ? What is it ?

An index is a data structure that organizes data records on persistent storage to optimize certain kinds of retrieval operations

An index is used to efficiently retrieve all records that satisfy a search condition on the search key fields of the index

An index is a function $f : K \rightarrow \wp(id_R)$ where K is set of keys and

$\wp(id_R)$ is a powerset (a set of all sets) of identifiers (addresses) id_R of the records in a set R

Let `EMP` be a relational table over a relational schema

`Employee(enumber, name, department)`

Then, $F_{department}domain(department) \rightarrow \wp(id_{EMP})$ is a function that maps the names of departments in domain(DEPARTMENT) into the sets of identifiers of rows $\wp(id_{EMP})$ in relational table `EMP`

$F_{department}$('Sales') returns the identifiers of all rows where a value of attribute `department` is equal to 'Sales'

Introduction to Indexing

Outline

Index versus indexed file organization

An indexed file organization (index organized file) is a function $f : K \rightarrow \wp(R)$ where K is a set of keys and $\wp(R)$ is a powerset (a set of sets) of records R

An index maps a value into set of row identifiers

An index organized file maps a value into a set of records

A relational table can be indexed or it can be index organized

An indexed relational table consists of several index(es) created separately from implementation of a relational table itself

An index organized relational table consists only of implementation of one index where an index key is the same as a relational schema of an index organized table

Indexing in database systems is transparent to data manipulation and data retrieval operations

It means that a database system automatically modifies an index and automatically decides whether an index is used for search

Introduction to Indexing

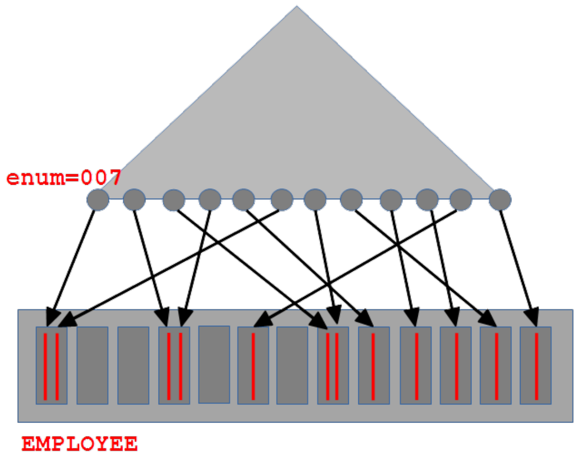
Outline

Primary (unique) index

A primary (unique) index is an index on a set of attributes equal to primary or candidate key

A primary index is a function $f : K \rightarrow id_R$ where K is a set of key values and id_R is a set of identifiers (physical addresses) of rows in a relational table R

A primary index maps and index key into a single row identifier (physical address of a row)



F_{enum}: domain(enum) $\rightarrow id_{EMPLOYEE}$

Primary (unique) index

A primary key in a relational table is always automatically indexed by a database system

For example, a relational table `EMPLOYEE` created over a relational schema `Employee(enum, name, department)` where `enum` is a primary key has an index automatically created on an attribute (`enum`)

For example, a relational table `ENROLMENT` created over a relational schema `Enrolment(snumber,code,edate)` where (`snumber,code)` is a primary key has an index automatically created on a set of attributes (`snumber,code`)

An index on (`snumber,code`) is a composite index

Introduction to Indexing

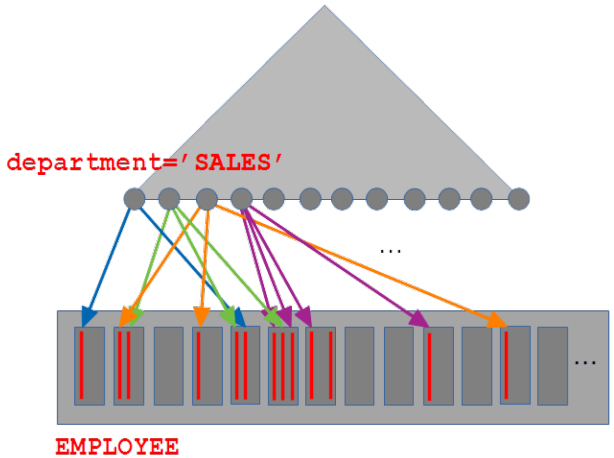
Outline

Secondary (nonunique) index

A secondary index is an index which is not primary

A secondary index is a function $f : K \rightarrow \wp(R)$ where K is a set of key values and id_R is a set of identifiers (physical addresses) of rows in a relational table R

A secondary index maps and index key into a single row identifier (physical address of a row)



$F_{department}$: domain(department) $\rightarrow \wp(id_{EMPLOYEE})$

Secondary (nonunique) index

For example, an index on an attrtibute (`name`) in a relational table `EMPLOYEE` created over a relational schema `Employee(enum, name, department)` is a secondary (nonunique) index

For example, an index on a set of attrtibutes (`name, department`) in a relational table `EMPLOYEE` created over a relational schema `Employee(enum, name, department)` is a secondary index

For example, an index on an attribute (`snumber`) in a relational table `ENROLMENT` created over a relational schema `Enrolment(snumber,code,edate)` is a secondary index

An index on a set of attributes (`enum,name`) in a relational table `EMPLOYEE` created over a relational schema `Employee(enum, name, department)` is still a primary index because (`enum,name`) is a superkey

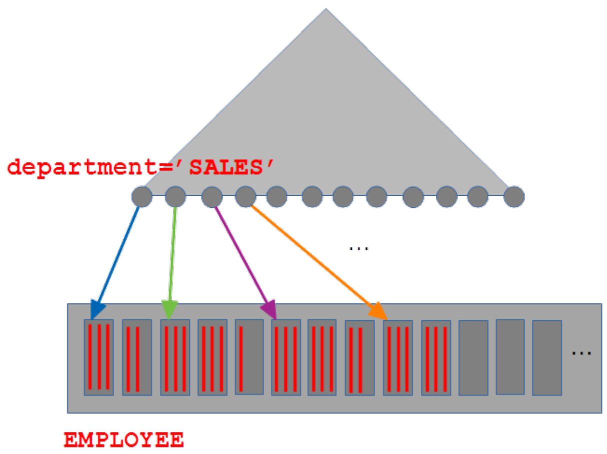
Introduction to Indexing

Outline

Clustered index

A clustered index is an index organized such that the ordering of rows is the same as ordering of keys in the index

A clustered index is a function $f: K \rightarrow id_R$ where K is a set of keys and id_R is a set of row identifiers (addresses) in a relational table R such that $f(v)$ returns row identifier (address) of the first row in a sequence of rows such that a value of attribute K is equal to v



$f_{department}$: domain(department) $\rightarrow id_{EMPLOYEE}$

Clustered index

Every primary index is clustered

Clustered index provides faster access to data than nonclustered secondary index

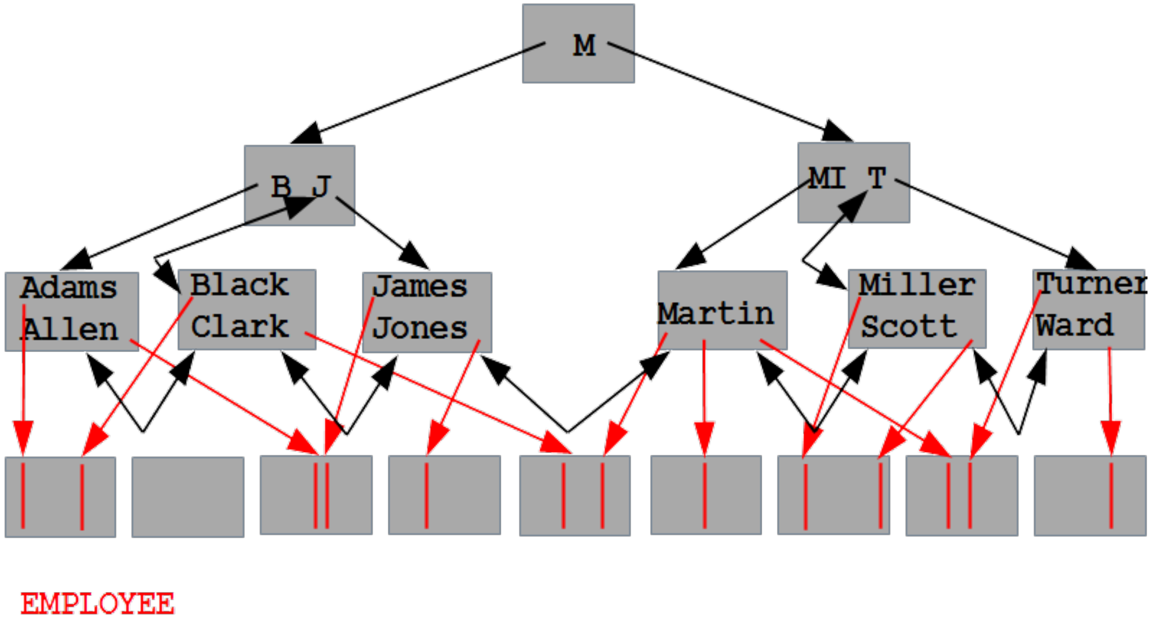
Clustered index has a very negative impact on performance of `INSERT` and `UPDATE` SQL statements

Therefore, clustered indexing should be applied to mainly to read-only data

Introduction to Indexing

Outline

B*-tree index implementation



B*-tree can be traversed either:

- vertically from root to leaf level of a tree
- horizontally either from left corner of leaf level to right corner of leaf level or the opposite
- vertically and later on horizontally either towards left lower corner or right lower corner of leaf level

B*-tree index implementation

The height of B*-tree is a length of path from root node to any of leaf nodes measured either in a number of nodes or edges along the path

The fanout of B*-tree is an average number of children nodes attached to a non-leaf node

If B*-tree has the height equal to h and the fanout equal to f then such a tree has f^h leaf nodes

If k is the total number of keys then $k = f^h$

Hence $\log_f(k) = h$

The height of a tree $h = \log_f(k) + 1$

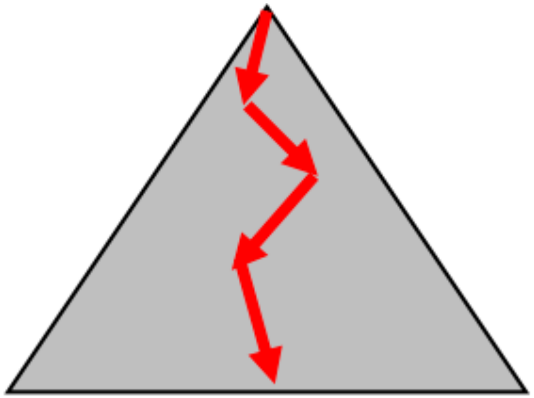
Introduction to Indexing

Outline

Traversals of B*-tree index

Vertical traversal

- Vertical traversal passes through B*-tree from the root node to one of the leaf nodes



- The total number of data blocks read during a vertical scan of B*-tree is equal to the height of B*-tree `h` measured in the number of nonleaf nodes `+ 1`
- The total number of data blocks read during a vertical traversal of B*-tree is equal to $\log_f(k) + 1$ where `f` is a fanout and `k` is the total number of keys included in the index

Traversals of B*-tree index

An index on a primary key (`enum`) in a relational table `EMPLOYEE` created over a relational schema `Employee(enum, name, department, salary)` is always built automatically by a database system

A name of an index is the same as a name of primary key constraint in a relational table `EMPLOYEE`

The following queries are processed through a vertical traversal of an index on (`enum`)

```sql
SELECT *
FROM EMPLOYEE
WHERE enum = 007;
```

```sql
SELECT *
FROM EMPLOYEE
WHERE enum = 007 AND department = 'MI6';
```

```sql
SELECT enum
FROM EMPLOYEE
WHERE enum = 007;
```

Traversals of B*-tree index

Assume that we created an index on attribute (`name`) in a relational table `EMPLOYEE` created over a relational schema `Employee(enum, name, department, salary)`

The following queries will be processed through a vertical traversal of an index on (`name`)

```sql
SELECT *
FROM EMPLOYEE
WHERE  name = 'James';
```

```sql
SELECT *
FROM EMPLOYEE
WHERE name = 'James' and department = 'MI6';
```

```sql
SELECT count(*)
FROM EMPLOYEE
WHERE name = 'James'
```

Traversals of B*-tree index

Assume that we created an index on the attributes (`name, department`) in a relational table `EMPLOYEE` created over a relational schema `Employee(enum, name, department, salary)`

The following queries are processed through a vertical traversal of an index on (`name, department`)

```sql
SELECT *
FROM EMPLOYEE
WHERE name = 'James' and department = 'MI6';
```

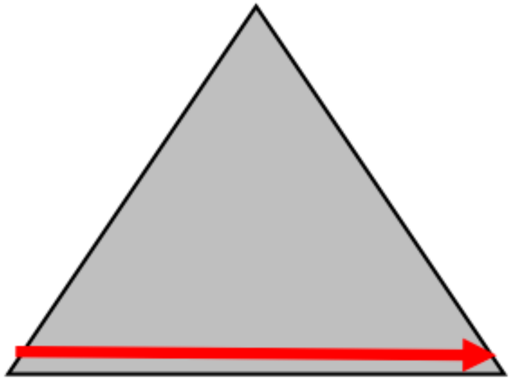
```sql
SELECT count(*)
FROM EMPLOYEE
WHERE name = 'James' and department = 'MI6';
```

```sql
SELECT *
FROM EMPLOYEE
WHERE name = 'James' and department = 'MI6' and salary > 1000;
```

Traversals of B*-tree index

Horizontal traversal

- Horizontal traveral passes through B*-tree from the leftmost (rightmost) leaf level node to the rightmost (leftmost) leaf level node



- The total number of data blocks read during a horizontal traversal of B*-tree is equal to the total number of leaf level nodes `w`
- The height of B*-Tree and the total number of leaf level nodes may be obtained from a data dictionary view `USER_TABLES` after processing of `ANALYZE TABLE` statement

Traversals of B*-tree index

The following queries are processed through a horizontal traversal of leaf level of an index on (`enum`)

```sql
SELECT COUNT(*)
FROM EMPLOYEE;
```

```sql
SELECT COUNT(enum)
FROM EMPLOYEE;
```

```sql
SELECT COUNT(name)  /* Only if name IS NOT NULL */
FROM EMPLOYEE;
```

```sql
SELECT enum
FROM EMPLOYEE;
```

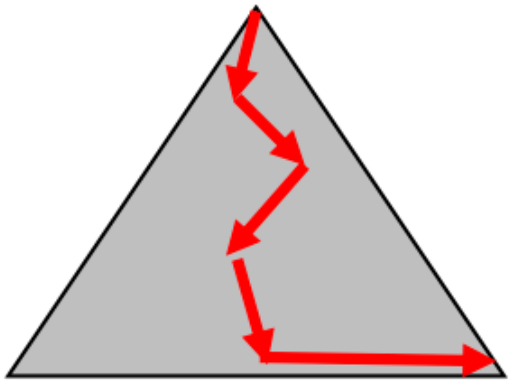
```sql
SELECT enum, COUNT(*)
FROM EMPLOYEE
GROUP BY enum;
```

```sql
SELECT enum
FROM EMPLOYEE
ORDER BY enum;
```

Traversals of B*-tree index

Vertical and horizontal traversal

- Vertical and horizontal traversal passes through B*-tree from the root node to one of the leaf nodes and later on the leaf nodes towards the rightmost (leftmost) leaf node



- The total number of data blocks read during a vertical and horizontal traversal of B*-tree is equal to $\log_f(k) + 1 + p*w$ where f is a fanout, k is the total number of keys, w is the total number of leaf level blocks and p is a fraction of w to be traversed

Traversals of B*-tree index

The following queries can be processed through a vertical traversal and later on horizontal traversal of an index on (`enum`)

```sql
SELECT *
FROM EMPLOYEE
WHERE enum > 300;
```

```sql
SELECT count(*)
FROM EMPLOYEE
WHERE enum < 007;
```

```sql
SELECT *
FROM EMPLOYEE
WHERE enum > 300 and salary > 1000;
```

Traversals of B*-tree index

Assume that we created an index on the attributes (`name, department`) in a relational table `EMPLOYEE` created over a relational schema `Employee(enum, name, department, salary)`

The following queries can be processed through a vertical traversal and later on horizontal traversal of an index on (`name, department`)

```sql
SELECT *
FROM EMPLOYEE
WHERE name > 'James';
```

```sql
SELECT count(*)
FROM EMPLOYEE
WHERE name <= 'James';
```

```sql
SELECT *
FROM EMPLOYEE
WHERE name = 'James' and department > 'MI6';
```

Traversals of B*-tree index

Assume that we created a composite key index on the attributes (`name, department`) in a relational table `EMPLOYEE` created over a relational schema `Employee(enum, name, department, salary)`

The following queries can be processed through a vertical traversal and later on horizontal traversal of an index on (`name, department`)

```sql
SELECT *
FROM EMPLOYEE
WHERE name > 'James' and salary > 1000;
```

```sql
SELECT name, count(*)
FROM EMPLOYEE
WHERE name > 'James' and salary > 1000
GROUP BY name;
```

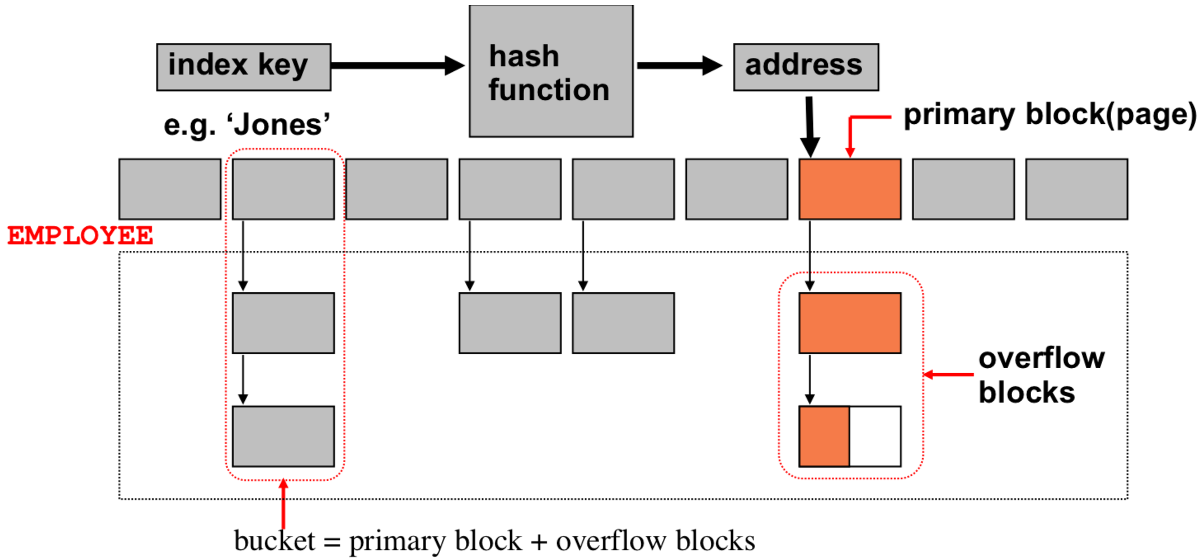
```sql
SELECT *
FROM EMPLOYEE
WHERE name > 'James' and salary > 1000
ORDER BY name;
```

Introduction to Indexing

Outline

Hash-based index implementation

A hash-based index consists of a sequence of primary data blocks and the "buckets" of overflow data blocks linked to the primary data blocks



To insert a row into hash-based index a hash function is computed over a key

A result of hash function determines a primary block or a "bucket" of overflow blocks that should be used for insertion

Hash-based index implementation

A hash function has the following properties

- A hash function `h` does not have an inverse function, for some keys $k_1, k_2, \ldots, k_n$ $h(k_1) = h(k_2) = \ldots = h(k_n)$
- A result of hash function determines a primary block or a "bucket" of overflow blocks that should be used for insertion
- Quality of hashing depends on how evenly a hash function distributes the records over buckets
- The best quality is achieved if size of each bucket is more or less the same
- A sample hash function: (folding and xor-ing n bits) mod number-of-buckets

References

Elmasri R. and Navathe S. B., Fundamentals of Database Systems, Chapter 17 Indexing Structures for Files and Physical Database Design, 7th ed., The Person Education Ltd, 2017