SQL Indexing

COSC 404 – Database System Implementation

Creating Indexes in SQL

There are two general ways of creating an index:

- 1) By specifying it in your `CREATE TABLE` statement:

```
CREATE TABLE test
(    a int,
     b int,
     c varchar(10)
     PRIMARY KEY (a),
     UNIQUE (b),
     INDEX (c)
);
```

Only one primary key index allowed.

`UNIQUE` index does not allow duplicate keys.

Creates an index that supports duplicates.

- 2) Using a `CREATE INDEX` command after a table is created:

```
CREATE INDEX myIdxName ON test (a,b);
```

CREATE INDEX Command

CREATE INDEX syntax:

```
CREATE [UNIQUE] INDEX indexName
    ON  tableName (colName [ASC|DESC] [,...])

DROP INDEX indexName;
```

- UNIQUE means that each value in the index is unique.
- ASC/DESC specifies the sorted order of index.
- Note: The syntax varies slightly between systems.

CREATE INDEX Command Examples

Examples:

```
CREATE UNIQUE INDEX idxStudent ON Student(sid)
```

- Creates an index on the field `sid` in the table `Student`
- `idxStudent` is the name of the index.
- The `UNIQUE` keyword ensures the uniqueness of `sid` values in the table (and index).
  - Uniqueness is enforced even when adding an index to a table with existing data. If the `sid` field is non-unique then the index creation fails.

```
CREATE INDEX clMajor ON Student(Major) CLUSTER
```

- Creates a clustered (primary) index on the `Major` field of `Student` table.
- Note: Clustered index may or may not be on a key field.

`CREATE INDEX` Command Examples (2)

```
CREATE INDEX idxMajorYear ON student(Major,Year)
```

- Creates an index with two fields.
- Duplicate search keys are possible.

Creating Indexes in MySQL

MySQL supports both ways of creating indexes. The `CREATE INDEX` command is mapped to an `ALTER TABLE` statement.

Syntax for `CREATE TABLE`:

```
CREATE TABLE tbl_Name
(
    [CONSTRAINT [name]] PRIMARY KEY [index_type]
(index_col,...)
  | KEY [index_name] [index_type] (index_col,...)
  | INDEX [index_name] [index_type] (index_col,...)
  | [CONSTRAINT [symbol]] UNIQUE [INDEX]
        [index_name] [index_type] (index_col,...)
  | [FULLTEXT|SPATIAL] [INDEX] [index_name] (index_col,...)
  | [CONSTRAINT [symbol]] FOREIGN KEY
        [index_name] (index_col_name,...)
    ...
)
```

Creating Indexes in MySQL (2)

Notes:

- 1) By specifying a primary key, an index is automatically created by MySQL. You do not have to create another one!
- 2) The primary key index (and any other type of index) can have more than one attribute.
- 3) MySQL assigns default names to indexes if you do not provide them.
- 4) MySQL supports B+-tree, Hash, and R-tree indexes but support depends on table type.
- 5) Can index only the first few characters of a `CHAR`/`VARCHAR` field by using col_name(length) syntax. (smaller index size)
- 6) FULLTEXT indexes allow more powerful natural language searching on text fields (but have a performance penalty).
- 7) SPATIAL indexes can index spatial data.

Creating Indexes in SQL Server

Microsoft SQL Server supports defining indexes in the `CREATE TABLE` statement or using a `CREATE INDEX` command.

Notes:

- 1) The primary index is a cluster index (rows sorted and stored by indexed column). Unique indexes are non-clustered.
  - A clustered (primary) index stores the records in the index.
  - A secondary index stores pointers to the records in the index.
  - Clustered indexes use B+-trees.
- 2) A primary key constraint auto-creates a clustered index.
- 2) Also supports full-text and spatial indexing.

Performance Improvement of Indexes

Indexes can improve query performance, especially when indexing foreign keys ***and*** for queries with ***low selectivity***.

Experiment:

- Use TPC-H database and perform join between `Orders` and `Customer` where the `o_custkey` field in `Orders` table is and is not indexed.
- select * from orders o, customers c where o.o_custkey = c.c_custkey
  - Result size = 1,500,000 rows in time 40 seconds
- add condition: `where o_custkey = 10`
  - # of rows = 20, without index = 7 seconds ; with index = less than a second
- add condition: `where o_custkey < 100`
  - # of rows = 979; without index = 7 seconds; with index = less than a second
- add condition: `where o_custkey < 1000`
  - What do you think will be faster a) with or b) without an index?

Bottom line: Indexes improve performance but only for queries that have low selectivity (get return rows from index).

Indexing with Multiple Fields

Consider an index with multiple fields:

```
CREATE INDEX idxMajorYear ON student(Major,Year)
```

and a query that could use this index:

```
SELECT * FROM student WHERE Major="CS" and Year="3"
```

Commercial databases use a B+-tree index. Note order is important as the index is sorted on the attributes in order.

There are also other methods for multiple field indexing:

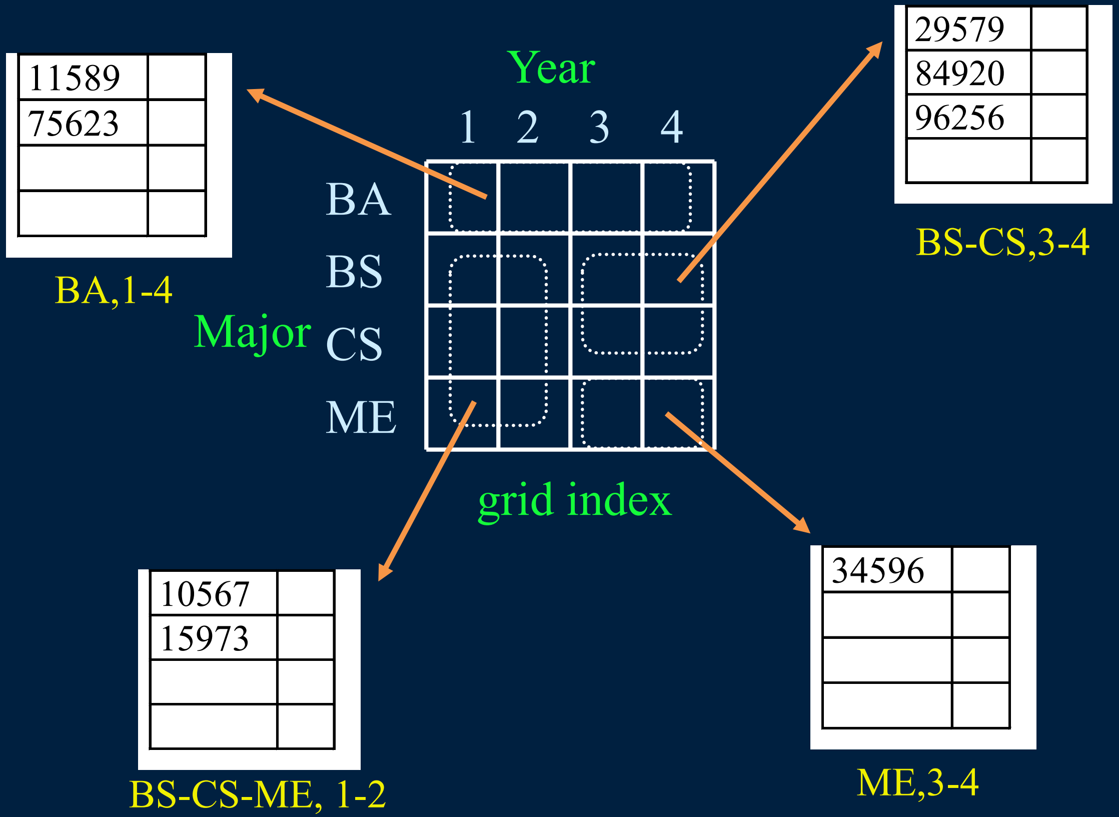
- Partitioned Hashing
- Grid Files

Multiple Key Indexing
Grid Files

A ***grid file*** is designed for multiple search-key queries.

- The grid file has a grid array and a linear scale for each search-key attribute.
- The grid array has a number of dimensions equal to number of search-key attributes.
- Each cell of the grid points to a disk bucket. Multiple cells of the grid array can point to the same bucket.
- To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer.
- If a bucket becomes full, a new bucket can be created if more than one cell points to it. If only one cell points to it, an overflow bucket needs to be created.

Example Grid File for Student Database



Grid Files Querying

A grid file on two attributes A and B can answer queries:

- Exact match queries:
  - A=value
  - B=value
  - A=value AND B=value
- Range queries:
  - $(a_1 \leq A \leq a_2)$
  - $(b_1 \leq B \leq b_2)$
  - $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2)$
- For example, to answer $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2)$, use linear scales to find candidate grid array cells, and look up all the buckets pointed to from those cells.

Linear scales must be chosen to uniformly distribute records across cells. Otherwise there will be many overflow buckets.

Grid Files Discussion

Using grid cells as bucket pointers allows the grid to be regular, but increases the indirection.

Note that the linear scales are often allocated in a table where each value maps to a number between 0 and ***N***.

This allows easier indexing of the grid, and also permits the linear scales to be ranges. Example:

Salary Linear Scale

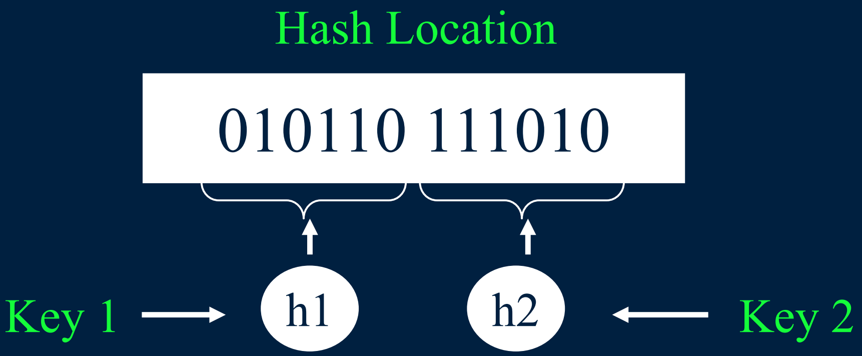
$0-$10,000	0
$10,001-$50,000	1
$50,001-$100,000	2
$100,000+	3

Overall: Grid files are good for multi-key searches but require space overhead and ranges that evenly split keys.

★ Multiple Key Indexing
Partitioned Hashing

The idea behind ***partitioned hashing*** is that the overall hash location is a combination of the hash values from each key.

For example,



The overall hash location L is 12 bits long.
The first 6 bits are from $h1$, the second 6 from $h2$.

Partitioned Hashing Example

Hash Table

h1 is hash function for Major.
h1(BA) = 0
h1(BS)=0
h1(CS)=1
h1(ME)=1
….

h2 is hash function for Year.
h2(1) = 00
h2(2) = 01
h2(3) = 10
h2(4) = 11
….

Hash Table

000	29579
001	11589
010	75623
011	
100	
101	96256
110	10567,15973
111	34596,84920

<10567,CS,3>, <11589,BA,2>, <15973,CS,3>, <29579,BS,1>,<34596,ME,4>, <75623,BA,3>, <84920,CS,4>, <96256,ME,2>

Partitioned Hashing Example Searching

Hash Table

h1 is hash function for Major.

h1(BA) = 0

h1(BS)=0

h1(CS)=1

h1(ME)=1

….

h2 is hash function for Year.

h2(1) = 00

h2(2) = 01

h2(3) = 10

h2(4) = 11

….

Hash Table

000	29579
001	11589
010	75623
011	
100	
101	96256
110	10567,15973
111	34596,84920

Major="CS" AND Year="3"

Partitioned Hashing Example Searching (2)

Hash Table

h1 is hash function for Major.

h1(BA) = 0

h1(BS)=0

h1(CS)=1

h1(ME)=1

….

h2 is hash function for Year.

h2(1) = 00

h2(2) = 01

h2(3) = 10

h2(4) = 11

….

Year="2"

Hash Table

000	29579
001	11589
010	75623
011	
100	
101	96256
110	10567,15973
111	34596,84920

Partitioned Hashing Example Searching (3)

Hash Table
h1 is hash function for Major.
h1(BA) = 0
h1(BS)=0
h1(CS)=1
h1(ME)=1
….

h2 is hash function for Year.
h2(1) = 00
h2(2) = 01
h2(3) = 10
h2(4) = 11
….

Hash Table

000	29579
001	11589
010	75623
011	
100	
101	96256
110	10567,15973
111	34596,84920

Major="BA"

Partitioned Hashing Question

Hash Table

h1 is hash function for Major.

h1(BA) = 0

h1(BS)=0

h1(CS)=1

h1(ME)=1

....

h2 is hash function for Year.

h2(1) = 00

h2(2) = 01

h2(3) = 10

h2(4) = 11

....

Major="BS" OR Year="1"

Buckets searched:

A) 2 buckets

B) 4 buckets

C) 5 buckets

D) 6 buckets

E) 8 buckets

Grid Files versus Partitioned Hashing

Both grid files and partitioned hashing have different query performance.

Grid Files:

- Good for all types of queries including range and nearest-neighbor queries.
- However, many buckets will be empty or nearly empty because of attribute correlation. Thus, grid can be space inefficient.

Partitioned Hashing:

- Useless for range and nearest-neighbor queries because physical distance between points is not reflected in closeness of buckets.
- However, hash function will randomize record locations which should more evenly divide records across buckets.
  - Partial key searches should be faster than grid files.

Bitmap Indexes

A ***bitmap index*** is useful for indexing attributes that have a small number of values. (e.g. gender)

- For each attribute value, create a bitmap where a 1 indicates that a record at that position has that attribute value.
- Retrieve matching records by id.

student table

Rec#	St. ID	Name	Mjr	Yr
0	10567	J. Doe	CS	3
1	11589	T. Allen	BA	2
2	15973	M. Smith	CS	3
3	29579	B. Zimmer	BS	1
4	34596	T. Atkins	ME	4
5	75623	J. Wong	BA	3
6	84920	S. Allen	CS	4
7	96256	P. Wright	ME	2

bitmap index on Mjr

Mjr	bitmap
BA	01000100
BS	00010000
CS	10100010
ME	00001001

bitmap index on Yr

Yr	bitmap
1	00010000
2	01000001
3	10100100
4	00001010

How could we use bitmap indexes to answer:

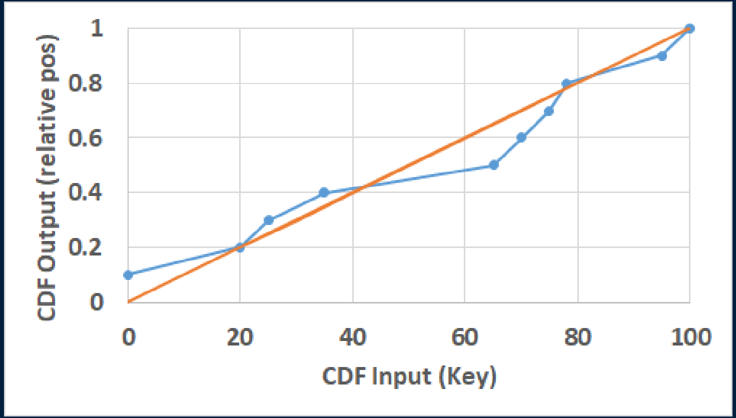
```
SELECT count(*) FROM student
WHERE Mjr = 'BA' and Year=2
```

Index Research – Learned Indexes

A ***learned index*** uses machine learning to predict the location of a key in sorted data by estimating the cumulative density function (CDF).

- There have been various models proposed including recursive model index (RMI), piecewise geometric models (PGM), linear regressions, and radix spline.

Value	Relative Position	y=x/100	Error
0	0.1	0	0.1
20	0.2	0.2	0
25	0.3	0.25	0.05
35	0.4	0.35	0.05
65	**0.5**	**0.65**	**0.15**
70	0.6	0.7	0.1
75	0.7	0.75	0.05
78	0.8	0.78	0.02
95	0.9	0.95	0.05
100	1	1	0



Example: Use simple linear model y = x/100. Maximum error at value 65 of 0.15.
Predict location of key 35.
Search range is: (10*(35/100 - .15), 10*(35/100+.15)) = **(2,5)**

Conclusion

The index structures we have seen, specifically, B+-trees are used for indexing in commercial database systems.

- There are also special indexing structures for text and spatial data.

When tuning a database, examine the types of indexes you can use and the configuration options available.

Grid files and ***partitioned hashing*** are specialized indexing methods for multi-key indexes.

Bitmap indexes allow fast lookups when attributes have few values and can be efficiently combined using logical operations.

Major Objectives

The "One Things":

- Use index structures in SQL using CREATE/ALTER commands.
- Perform insertions and searches using partitioned hashing.

Major Theme:

- Various DBMSs give you control over the types of indexes that you can use and the ability to tune their parameters. Knowledge of the underlying index structures helps performance tuning.

Objectives:

- Perform searches using grid files.
- Understand how bitmap indexes are used for searching and why they provide a space and speed improvement in certain cases.

THE UNIVERSITY OF BRITISH COLUMBIA