Practice and Applications of Data Management

CMPSCI 345

Lecture 09: Indexes and Tuning
Indexes or indices

- Primary mechanism to make queries run faster

- Index on attribute R.A:
  - Creates additional persistent data structure stored with the database
  - Can dramatically speed up certain operations:
    - Find all R tuples where R.A = v
    - Find all R and S tuples where R.A = S.B
    - Find all R tuples where R.A > v (sometimes, depending on index type)
## High-level overview: indexes

<table>
<thead>
<tr>
<th>id</th>
<th>age</th>
<th>salary</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>006</td>
<td>19</td>
<td>50k</td>
<td>...</td>
</tr>
<tr>
<td>005</td>
<td>20</td>
<td>55k</td>
<td>...</td>
</tr>
<tr>
<td>004</td>
<td>25</td>
<td>50k</td>
<td>...</td>
</tr>
<tr>
<td>007</td>
<td>30</td>
<td>80k</td>
<td>...</td>
</tr>
<tr>
<td>002</td>
<td>35</td>
<td>75k</td>
<td>...</td>
</tr>
<tr>
<td>003</td>
<td>35</td>
<td>70k</td>
<td>...</td>
</tr>
<tr>
<td>001</td>
<td>40</td>
<td>65k</td>
<td>...</td>
</tr>
</tbody>
</table>

**data file = index file**

**clustered (primary) index**

**index file**

**unclustered (secondary) index**
Database file types

The data file can be one of:

- **Heap file**
  - Set of records, partitioned into blocks
  - Unsorted

- **Sequential file**
  - Sorted according to some attribute(s) called *(sort)* key

  different from “key”!
Index

- A (possibly separate) file, that allows fast access to records in the data file given a search key
- The index contains (key, value) pairs:
  - The key = an attribute value
  - The value = either a pointer to the record, or the record itself

again different from “key”!
Index classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
  - Unclustered = records close in index may be far in data

- **Primary/secondary**
  - Primary = is over attributes that include the primary key
  - Secondary = otherwise

- Organization: B+ tree or Hash table
Clustered/Unclustered

- **Clustered**
  - Index determines the location of indexed records
  - Typically, clustered index is one where values are data records (but not necessary)

- **Unclustered**
  - Index cannot reorder data, does not determine data location
  - In these indexes: \textit{value} = pointer to data record

```sql
CLUSTER tableName USING indexName
```
Clustered index

- File is sorted on the index attribute
- Only one per table
Unclustered index

- Several per table

Diagram showing the relationship between index file and data file.
Clustered vs. unclustered index

More commonly, in a clustered B+ Tree index, data entries are data records
Hash-based index

Good for point queries but not range queries

h2(age) = 00

h2(age) = 01

unclustered index

clustered index

h1(sid) = 00

h1(sid) = 11
Hash Table v.s. B+ tree

- B-tree search:
  - $O(\log n)$
  - Range and equality queries

- Hash search:
  - $O(1)$
  - Equality only

Rule 1: always use a B+ tree 😊
Rule 2: use a Hash table on K when:
  - There is a very important selection query on equality (WHERE K=?), and no range queries
  - You know that the optimizer uses a nested loop join where K is the join attribute of the inner relation

```sql
CREATE INDEX indexName
ON tableName
USING hash(column)
```
Practice

- Start Postgres and connect to your IMDB database

- Type: `\timing on`
  - Now postgres will report the running time for your queries

- Check for any existing indexes: `\di`
  - Postgres automatically creates indexes on primary keys
Practice

```sql
SELECT * FROM Actor WHERE lname = 'Bacon'
```

How long does it take to run?

Let’s see how the query is executed:

```sql
EXPLAIN
SELECT * FROM Actor WHERE lname = 'Bacon'
```
Introduce indexes

CREATE INDEX actorLName ON Actor(lname)

SELECT * FROM Actor WHERE lname = 'Bacon'

How long does it take now?

Let’s see how the query is executed this time:

EXPLAIN
SELECT * FROM Actor WHERE lname = 'Bacon'
Practice

Look at the indexes on table Actor: `\d Actor`

Let’s get execution plans for different queries:

```
EXPLAIN
SELECT  *
FROM    Actor
WHERE   lname = 'Bacon' AND id > 50000
```

```
EXPLAIN
SELECT  *
FROM    Actor
WHERE   lname = 'Bacon' AND id = 50000
```
Indexes and joins

```
SELECT  C.role
FROM    Actor A, Casts C
WHERE   lname = 'Bacon' AND A.id = C.pid
```

How long does it take to run?

Let’s see how the query is executed:

```
EXPLAIN
SELECT  C.role
FROM    Actor A, Casts C
WHERE   lname = 'Bacon' AND A.id = C.pid
```
EXPLAIN

How the join happens

How ‘Casts’ is accessed

How ‘Actor’ is accessed
Indexes and joins

```sql
CREATE INDEX castActorId
ON Casts(pid)
```

```sql
SELECT C.role
FROM Actor A, Casts C
WHERE lname = 'Bacon' AND A.id = C.pid
```

How long does it take now?

Let’s see how the query is executed this time:

```sql
EXPLAIN
SELECT C.role
FROM Actor A, Casts C
WHERE lname = 'Bacon' AND A.id = C.pid
```
EXPLAIN

QUERY PLAN

---

Nested Loop  (cost=0.00..1272.60 rows=233 width=12)
  ->  Index Scan using actorlname on actor a  (cost=0.00..117.96 rows=38 width=4)
      Index Cond: ((lname)::text = 'Bacon'::text)
  ->  Index Scan using castactor on casts c  (cost=0.00..28.53 rows=186 width=16)
      Index Cond: (pid = a.id)

Both indexes are used

Different type of join
Disadvantages

- Indexes speed up queries
  - SELECT FROM WHERE

- But they usually slow down updates:
  - INSERT, DELETE, UPDATE
  - However some updates benefit from indexes

```
UPDATE R
SET A = 7
WHERE K = 55
```
The Database Tuning Problem

- We are given a workload description
  - List of queries and their frequencies
  - List of updates and their frequencies
  - Performance goals for each type of query

- Perform physical database design
  - Choice of indexes
  - Tuning the conceptual schema
    - Denormalization, vertical and horizontal partition
  - Query and transaction tuning
The Index Selection Problem

- Given a database schema (tables, attributes)
- Given a “query workload”:
  - Workload = a set of (query, frequency) pairs
  - The queries may be both SELECT and updates
  - Frequency = either a count, or a percentage

- Select a set of indexes that optimizes the workload

In general this is a very hard problem
Index selection decisions

- To index or not to index?
- Which key?
- Multiple keys?
- Clustered or unclustered?
- Hash or trees?
Index Selection: Which Search Key

- Make some attribute K a search key if the WHERE clause contains:
  - An exact match on K
  - A range predicate on K
  - A join on K
The Index Selection Problem 1

V(M, N, P)

Your workload is this

100,000 queries:

```
SELECT * FROM V WHERE N=?
```

100 queries:

```
SELECT * FROM V WHERE P=?
```

What indexes?
The Index Selection Problem 1

V(M, N, P)

Your workload is this

100,000 queries:

\[
\begin{align*}
&\text{SELECT} \quad * \\
&\text{FROM} \quad V \\
&\text{WHERE} \quad N=？
\end{align*}
\]

100 queries:

\[
\begin{align*}
&\text{SELECT} \quad * \\
&\text{FROM} \quad V \\
&\text{WHERE} \quad P=？
\end{align*}
\]

A: V(N) and V(P) (hash tables or B-trees)
The Index Selection Problem 2

Your workload is this

100,000 queries:

```
SELECT * 
FROM V 
WHERE N>? and N<?
```

100 queries:

```
SELECT * 
FROM V 
WHERE P=?
```

100,000 queries:

```
INSERT INTO V 
VALUES (?, ?, ?)
```

What indexes?
The Index Selection Problem 2

\( V(M, N, P) \)

Your workload is this

100,000 queries:

\[
\text{SELECT } * \text{ FROM } V \text{ WHERE } N>? \text{ and } N<?
\]

100 queries:

\[
\text{SELECT } * \text{ FROM } V \text{ WHERE } P=?
\]

100,000 queries:

\[
\text{INSERT INTO } V \text{ VALUES (? , ? , ?)}
\]

A: definitely \( V(N) \) must B-tree; unsure about \( V(P) \)
The Index Selection Problem 3

V(M, N, P)

Your workload is this

100,000 queries:

```
SELECT * FROM V WHERE N=?
```

1,000,000 queries:

```
SELECT * FROM V WHERE N=? and P>?
```

100,000 queries:

```
INSERT INTO V VALUES (?, ?, ?)
```

What indexes?
The Index Selection Problem 3

V(M, N, P)

Your workload is this

100,000 queries:

\[
\text{SELECT} \quad \ast \\
\text{FROM} \quad V \\
\text{WHERE} \quad N=\text{?}
\]

1,000,000 queries:

\[
\text{SELECT} \quad \ast \\
\text{FROM} \quad V \\
\text{WHERE} \quad N=\text{?} \\
\text{and} \quad P>\text{?}
\]

100,000 queries:

\[
\text{INSERT INTO} \quad V \\
\text{VALUES} \quad (?, \text{?}, \text{?})
\]

A: V(N, P)
The Index Selection Problem 2

V(M, N, P)

Your workload is this

1,000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100,000 queries:

```
SELECT *
FROM V
WHERE P>? and P<?
```

What indexes?
The Index Selection Problem 2

\[ V(M, N, P) \]

Your workload is this

1,000 queries:

\[
\text{SELECT} \quad * \\
\text{FROM} \quad V \\
\text{WHERE} \quad N>? \quad \text{and} \quad N<?
\]

100,000 queries:

\[
\text{SELECT} \quad * \\
\text{FROM} \quad V \\
\text{WHERE} \quad P>? \quad \text{and} \quad P<?
\]

A: \( V(N) \) unclustered; \( V(P) \) clustered
Basic Index Selection Guidelines

- Consider queries in workload in order of importance

- Consider relations accessed by query
  - No point indexing other relations

- Look at WHERE clause for possible search key

- Try to choose indexes that speed-up multiple queries

- And then consider the following...
Consider creating a multi-attribute key on K1, K2, if ...

WHERE clause has matches on K1, K2, ...

But also consider separate indexes

SELECT clause contains only K1, K2, ..

A covering index is one that can be used exclusively to answer a query, e.g. index R(K1,K2) covers the query:

```
SELECT K2
FROM R
WHERE K1=55
```

Can be answered with an index-only plan
To Cluster or Not to Cluster?

- Range queries benefit mostly from clustering

- Covering indexes do *not* need to be clustered
  
  Why?
SELECT * FROM R WHERE K>? and K<?