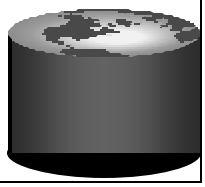
File Organizations and Indexing

Lecture 5
R&G Chapter 8

"If you don't find it in the index, look very carefully through the entire catalogue."

-- Sears, Roebuck, and Co., Consumer's Guide, 1897



Administrivia

- **Homework 1 Assignment now Available**
  - Due in 2 parts: Sunday 9/14, Thursday 9/18
  - Have to make changes to Postgres
  - Don't procrastinate

Review – The Big Picture

- **Databases are cool**
  - Theory: data modelling, relational algebra
  - Practical: storing data with disk, RAM
- **Today:**
  - Practical: File Organization, Indexing
- **Next Time:**
  - Theory: Relational Calculus

Review – Relational Algebra

- **Basic operators for manipulating relations**
  - *Selection* (σ) Selects a subset of *rows*
  - *Projection* (π) Retains only wanted *columns*
  - *Cross-product* ($\times$) Combine two relations
  - *Set-difference* (—) Tuples in r1, but not in r2
  - *Union* ($\cup$) Tuples in r1 and/or in r2
  - *Rename* (?) Change column name
- **Other Useful Operators**
  - *Intersection* (Ç) Tuples in both relations
  - *Join* ($\bowtie$) Match common columns
  - *Division* (/) Tuples in A that match all in B

Relation Algebra 2: Division

- **Division A / B,**
  - columns in A must be a superset of columns in B
  - let C_{both} be the columns in both A and B
  - let $C_{A\text{-}Only}$ be the columns that are only in A
  - result has columns $C_{A\text{-}Only}$
  - result has only rows where values in $C_{A\text{-}Only}$ have a match with every value in B
  - can be defined using other operators

Review: Examples of Division A/B

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

A

pno
p2

B1

pno
p2
p4

B2

pno
p1
p2
p4

B3

sno
s1
s2
s3
s4

A/B1

sno
s1
s4

A/B2

sno
s1

A/B3

Review:

Find the names of Sailors who have reserved all boats

Reserves

sid	bid	day
22	101	10/10/96
58	103	11/12/96

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Boats

bid	bname	color
101	Interlake	Blue
102	Interlake	Red
103	Clipper	Green
104	Marine	Red

Find the names of sailors who've reserved all boats

- **Uses division; schemas of the input relations to / must be carefully chosen:**

$$r\ (Tempsids, (\pi_{sid,bid} Reserves) / (\pi_{bid} Boats))$$

$$\pi_{sname} (Tempsids \bowtie Sailors)$$

- To find sailors who've reserved all 'Interlake' boats:

$$..... / \pi_{bid}(\sigma_{bname='Interlake'} Boats)$$

Review – Buffer Management and Files

- **Storage of Data**
  - Fields, either fixed or variable length…
  - Stored in Records…
  - Stored in Pages…
  - Stored in Files
- **If data won't fit in RAM, store on Disk**
  - Need Buffer Pool to hold pages in RAM
  - Different strategies decide what to keep in pool

Today: File Organization

- **How to keep pages of records on disk**
- ***but* must support operations:**
  - scan all records
  - search for a record id "RID"
  - search for record(s) with certain values
  - insert new records
  - delete old records

Alternative File Organizations

Many alternatives exist, tradeoffs for *each:*

- Heap files:
  - Suitable when typical access is file scan of all records.
- Sorted Files:
  - Best for retrieval in *search key* order
  - Also good for search based on *search key*
- Indexes: Organize records via trees or hashing.
  - Like sorted files, speed up searches for *search key* fields
  - Updates are much faster than in sorted files.

Indexes

- **Sometimes need to retrieve records by the *values in one or more fields*, e.g.,**
  - Find all students in the "CS" department
  - Find all students with a gpa > 3
- **An *index* on a file is a:**
  - Disk-based data structure
  - Speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be index search key
  - *Search key* is not the same as *key*
    - (e.g. doesn't have to be unique ID).
- **An index**
  - Contains a collection of *data entries*
  - Supports efficient retrieval of all records with a given search key value **k**.

First Question to Ask About Indexes

- **What kinds of selections do they support?**
  - Selections of form field <op> constant
  - Equality selections (op is =)
  - Range selections (op is one of <, >, <=, >=, BETWEEN)
  - More exotic selections:
    - 2-dimensional ranges ("east of Berkeley and west of Truckee and North of Fresno and South of Eureka")
      - Or n-dimensional
    - 2-dimensional distances ("within 2 miles of Soda Hall")
      - Or n-dimensional
    - Ranking queries ("10 restaurants closest to VLSB")
    - Regular expression matches, genome string matches, etc.
    - One common n-dimensional index: R-tree
      - Supported in Oracle and Informix
      - See http://gist.cs.berkeley.edu for research on this topic

Index Classification

- **What selections does it support**
- **Representation of data entries in index**
  - what info is the index storing? 3 alternatives:
    - Data record with key value **k**
    - <**k**, rid of data record with search key value **k**>
    - <**k**, list of rids of data records with search key **k**>
- **Clustered vs. Unclustered Indexes**
- **Single Key vs. Composite Indexes**
- **Tree-based, hash-based, other**

Alternatives for Data Entry **k*** in Index

- **Three alternatives:**
  - Actual data record (with key value **k**)
  - <**k**, rid of matching data record>
  - <**k**, list of rids of matching data records>
- **Choice is orthogonal to the indexing technique.**
  - techniques: B+ trees, hash-tables, R trees, ...
  - Typically, index contains auxiliary information that directs searches to the desired data entries
- **Can have multiple (different) indexes per file.**
  - E.g. file sorted by *age*, with a hash index on *salary* and a B+tree index on *name*.

Alternatives for Data Entries (Contd.)

- **Alternative 1:**
  Actual data record (with key value k**)**
  - Index structure *is* file organization for data records (like Heap files or sorted files).
  - At most one index on a table can use Alternative 1.
  - Saves pointer lookups
  - Can be expensive to maintain with insertions and deletions.

Alternatives for Data Entries (Contd.)

Alternative 2
<k, rid of matching data record>
and Alternative 3
<k, list of rids of matching data records>

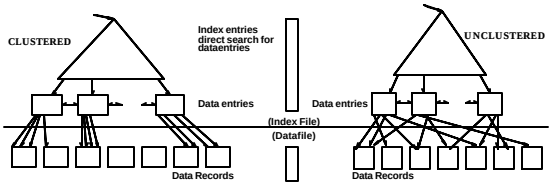
- Easier to maintain than Alt 1.
- At most one index can use Alternative 1; any others must use Alternatives 2 or 3.
- Alternative 3 more compact than Alternative 2, but leads to *variable sized data* entries even if search keys are of fixed length.
- Even worse, for large rid lists the data entry might have to span multiple pages!

Index Classification

- ***Clustered* vs. *unclustered*:**
  - If order of data records is the same as, or `close to', order of index data entries, then called *clustered index.*
- **A file can be clustered on at most one search key.**
- **Cost to retrieve data records with index varies *greatly* based on whether index clustered or not!**
- **Alternative 1 implies clustered, *but not vice-versa.***

Clustered vs. Unclustered Index

- **Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.**
  - To build clustered index, first sort the Heap file (with some free space on each block for future inserts).
  - Overflow blocks may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



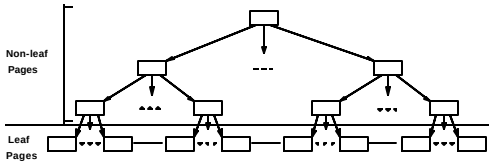
Unclustered vs. Clustered Indexes

- **What are the tradeoffs????**
- **Clustered Pros**
  - Efficient for range searches
  - May be able to do some types of compression
  - Possible locality benefits (related data?)
- **Clustered Cons**
  - Expensive to maintain (on the fly or sloppy with reorganization)

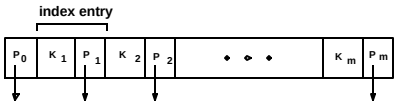
Hash-Based Indexes

- **Good for equality selections.**
  - Index is a collection of *buckets*. Bucket = *primary* page plus zero or more *overflow* pages.
  - *Hashing function* **h**:
    - **h**(*r*) = bucket in which record *r* belongs.
    - **h** looks at the *search key* fields of *r*.
- **If Alternative (1) is used, the buckets contain the data records; otherwise, they contain <key, rid> or <key, rid-list> pairs.**

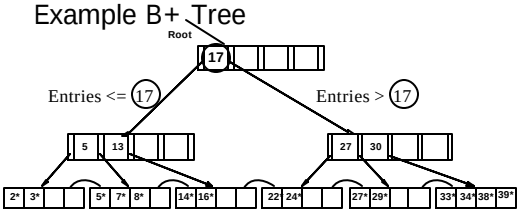
B+ Tree Indexes



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages contain *index entries* and direct searches:



Example B+ Tree



- **Find 28*? 29*? All > 15* and < 30***
- **Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.**
  - And change sometimes bubbles up the tree

Comparing File Organizations

- **Heap files (random order; insert at eof)**
- **Sorted files, sorted on *<age, sal>***
- **Clustered B+ tree file, Alternative (1), search key *<age, sal>***
- **Heap file with unclustered B + tree index on search key *<age, sal>***
- **Heap file with unclustered hash index on search key *<age, sal>***

Operations to Compare

- **Scan: Fetch all records from disk**
- **Fetch all records in sorted order**
- **Equality search**
- **Range selection**
- **Insert a record**
- **Delete a record**

Cost Model for Analysis

I/O cost 150,000 times more than hash function
- We ignore CPU costs, for simplicity

B: **The number of data pages**

R: **Number of records per page**

F: **Fanout of B-tree**

Average-case analysis; based on several simplistic assumptions.

✉ ***Good enough to show the overall trends!***

Assumptions in Our Analysis

- **Heap Files:**
  - Equality selection on key; exactly one match.
- **Sorted Files:**
  - Files compacted after deletions.
- **Indexes:**
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy => File size = 1.25 data size
  - Tree: 67% occupancy (this is typical).
    - Implies file size = 1.5 data size

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
S: Time required for equality search

	Heap File
Scan all records	
Get all in sort order	
Equality Search	
Range Search	
Insert	
Delete	

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
S: Time required for equality search

	Sorted File
Scan all records	
Get all in sort order	
Equality Search	
Range Search	
Insert	
Delete	

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
F: Fanout of B-Tree
S: Time required for equality search

	Clustered Tree
Scan all records	
Get all in sort order	
Equality Search	
Range Search	
Insert	
Delete	

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
F: Fanout of B-Tree
S: Time required for equality search

	Unclustered Tree
Scan all records	
Get all in sort order	
Equality Search	
Range Search	
Insert	
Delete	

I/O Cost of Operations

B: Number of data pages (packed)
R: Number of records per page
S: Time required for equality search

	Hash Index
Scan all records	
Get all in sort order	
Equality Search	
Range Search	
Insert	
Delete	

I/O Cost of Operations

B: The number of data pages
R: Number of records per page
F: Fanout of B-Tree
S: Time required for equality search

	Heap File	Sorted File	Clustered Tree	Unclustered Tree	Hash Index
Scan all records	B	B	1.5 B	1.5 B	1.25 B
Get all in sort order	4B	B	1.5 B	4B	4B
Equality Search	0.5 B	$\log_2$ B	$\log_F$ (1.5 B)	$\log_F$ (1.5 B) + 1	2
Range Search	B	S + # matching pages	S + # matching pages	S + # matching records	1.25 B
Insert	2	S + B	S + 1	S + 2	4
Delete	0.5B + 1	S + B	0.5B + 1	S + 2	S + 2

Index Selection Guidelines

- **Attributes in WHERE clause are candidates for index keys.**
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- **Multi-attribute search keys should be considered when a WHERE clause contains several conditions.**
  - Order of attributes is important for range queries.
  - Such indexes sometimes enable index-only strategies
    - For index-only strategies, clustering is not important!
- **Choose indexes that benefit as many queries as possible.**
- **Since only one index can be clustered per table, choose it based on important queries that would benefit the most from clustering.**

Examples of Clustered Indexes

- **B+ tree index on E.age can be used to get qualifying tuples.**
  - How selective is the condition?
  - Is the index clustered?
- **Consider the GROUP BY query.**
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!
- **Equality queries and duplicates:**
  - Clustering on *E.hobby* helps!

```
SELECT E.dno
FROM Emp E
WHERE E.age>40
```

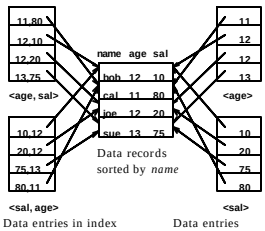
```
SELECT E.dno, COUNT(*)
FROM Emp E
WHERE E.age>10
GROUP BY E.dno
```

```
SELECT E.dno
FROM Emp E
WHERE E.hobby=Stamps
```

Indexes with Composite Search Keys

- ***Composite Search Keys*: Search on a combination of fields.**
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10
- **Data entries in index sorted by search key to support range queries.**
  - Lexicographic order, or
  - Spatial order.

Examples of composite key indexes using lexicographic order.



Composite Search Keys

- **To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.**
  - Choice of index key orthogonal to clustering etc.
- **If condition is: 20<*age*<30 AND 3000<*sal*<5000:**
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- **If condition is: *age*=30 AND 3000<*sal*<5000:**
  - Clustered <*age,sal*> index much better than <*sal,age*> index!
- **Composite indexes are larger, updated more often.**

Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

<*E.dno*>

```
SELECT D.mgr
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<*E.dno,E.eid*> *Tree index!*

```
SELECT D.mgr, E.eid
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<*E.dno*>

```
SELECT E.dno, COUNT(*)
FROM Emp E
GROUP BY E.dno
```

<*E.dno,E.sal*> *Tree index!*

```
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno
```

<*E. age,E.sal*> or <*E.sal, E.age*> *Tree!*

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
 E.sal BETWEEN 3000 AND 5000
```

Index-Only Plans (Contd.)

- **Index-only plans are possible if the key is <dno,age> or we have a tree index with key <age,dno>**
  - Which is better?
  - What if we consider the second query?

```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age=30
GROUP BY E.dno
```

```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age>30
GROUP BY E.dno
```

Summary

- **Alternative file organizations, tradeoffs for each**
- **If selection queries are frequent, sorting the file or building an *index* is important.**
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- **Index is a collection of data entries plus a way to quickly find entries with given key values.**

Summary (Contd.)

- **Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.**
  - Choice orthogonal to *indexing technique* used to locate data entries with a given key value.
- **Can have several indexes on a given file of data records, each with a different search key.**
- **Indexes can be**
  - clustered, unclustered
  - B-tree, hash table, etc.

Summary (Contd.)

- **Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.**
  - What are the important queries and updates? What attributes/relations are involved?
- **Indexes must be chosen to speed up important queries (and perhaps some updates!).**
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.