Lecture 7: Indexes and Database Tuning

Wednesday, November 10, 2010
The Take-home Final

- Poll: no date is good for everyone
- Will settle for maximum flexibility
- **Main take-home final**
  - December 4 and 5 (Saturday, Sunday)
  - Grades will be posted by December 11
- **Makeup take-home final**
  - Exact date TBD, but *before* December 9
  - On request (send me email)
A Note

Xquery replaced `document("...")` with `doc("...")`

- Slides have: `document("...")`
- You should use: `doc("...")`
Outline

• Storage and indexing: Chapter 8, 9, 10
  - Will start today, continue next week

• Database Tuning: Chapter 20
  - Will discuss today

• Security in SQL: Chapter 21
  - Will not discuss in class
Storage Model

• DBMS needs spatial and temporal control over storage
  - Spatial control for performance
  - Temporal control for correctness and performance

• For spatial control, two alternatives
  - Use “raw” disk device interface directly
  - Use OS files
Spatial Control
Using “Raw” Disk Device Interface

- **Overview**
  - DBMS issues low-level storage requests directly to disk device

- **Advantages**
  - DBMS can ensure that important queries access data sequentially
  - Can provide highest performance

- **Disadvantages**
  - Requires devoting entire disks to the DBMS
  - Reduces portability as low-level disk interfaces are OS specific
  - Many devices are in fact “virtual disk devices”
Spatial Control
Using OS Files

• Overview
  - DBMS creates one or more very large OS files

• Advantages
  - Allocating large file on empty disk can yield good physical locality

• Disadvantages
  - OS can limit file size to a single disk
  - OS can limit the number of open file descriptors
  - But these drawbacks have mostly been overcome by modern OSs
Commercial Systems

• Most commercial systems offer both alternatives
  - Raw device interface for peak performance
  - OS files more commonly used

• In both cases, we end-up with a DBMS file abstraction implemented on top of OS files or raw device interface
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 *key*

---

Note: “key” here means something else than “primary key”
Arranging Pages on Disk

- Block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder

- Blocks in a file should be arranged sequentially on disk (by `next`), to minimize seek and rotational delay.

- For a sequential scan, pre-fetching several pages at a time is a big win!
Representing Data Elements

- Relational database elements:

  ```
  CREATE TABLE Product (  
    pid INT PRIMARY KEY,  
    name CHAR(20),  
    description VARCHAR(200),  
    maker CHAR(10) REFERENCES Company(name)  
  )
  ```

- A tuple is represented as a record
- The table is a sequence of records
Issues

• Managing free blocks
• Represent the records inside the blocks
• Represent attributes inside the records
Managing Free Blocks

- Linked list of free blocks
- Or bit map
File Organization

Linked list of pages:

Header page

Data page Data page Data page

Full pages

Data page Data page Data page

Pages with some free space
File Organization

Better: directory of pages
Page Formats

Issues to consider
- 1 page = fixed size (e.g. 8KB)
- Records:
  - Fixed length
  - Variable length
- Record id = RID
  - Typically RID = (PageID, SlotNumber)

Why do we need RID’s in a relational DBMS?
Page Formats

Fixed-length records: packed representation

Rec 1  Rec 2  Rec N

Free space  N

Problems?
Page Formats

Variable-length records
Record Formats: Fixed Length

- Information about field types same for all records in a file; stored in system catalogs.
- Finding $i^{th}$ field requires scan of record.
- Note the importance of schema information!
Record Header

Need the header because:
- The schema may change for a while new+old may coexist
- Records from different relations may coexist
Variable Length Records

Other header information

header pid name descr maker

Place the fixed fields first: F1
Then the variable length fields: F2, F3, F4
Null values take 2 bytes only
Sometimes they take 0 bytes (when at the end)
BLOB

- Binary large objects
- Supported by modern database systems
- E.g. images, sounds, etc.
- Storage: attempt to cluster blocks together

CLOB = character large object
- Supports only restricted operations
File Organizations

- **Heap** (random order) files: Suitable when typical access is a file scan retrieving all records.
- **Sorted Files**: Best if records must be retrieved in some order, or only a `range` of records is needed.
- **Indexes**: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.
Modifications: Insertion

- File is unsorted: add it to the end (easy ^")
- File is sorted:
  - Is there space in the right block?
    - Yes: we are lucky, store it there
  - Is there space in a neighboring block?
    - Look 1-2 blocks to the left/right, shift records
  - If anything else fails, create overflow block
Modifications: Deletions

- Free space in block, shift records
- Maybe be able to eliminate an overflow block
- Can never really eliminate the record, because others may point to it
  - Place a tombstone instead (a NULL record)
Modifications: Updates

- If new record is shorter than previous, easy ^^
- If it is longer, need to shift records, create overflow blocks
Index

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

Note: “key” (aka “search key”) again means something else
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**
  - Meaning 1:
    - Primary = is over attributes that include the primary key
    - Secondary = otherwise
  - Meaning 2: means the same as clustered/unclustered

- **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: \texttt{value} = pointer to data record
Clustered Index

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

- Several per table
Clustered vs. Unclustered Index

B+ Tree

Data entries

(Data file)

Index File

Data Records

CLUSTERED

B+ Tree

Data entries

(Data file)

Index File

Data Records

UNCLUSTERED

Dan Suciu -- CSEP544 Fall 2010
Hash-Based Index

Good for point queries but not range queries

Another example of unclustered/secondary index

Another example of clustered/primary index
Alternatives for Data Entry \( k^* \) in Index

Three alternatives for \( k^* \):

- Data record with key value \( k \)

- \(<k, \text{rid} \text{ of data record with key } = k>\>

- \(<k, \text{list of rids of data records with key } = k>\)
Alternatives 2 and 3
B+ Trees

• Search trees

• Idea in B Trees
  - Make 1 node = 1 block
  - Keep tree balanced in height

• Idea in B+ Trees
  - Make leaves into a linked list: facilitates range queries

Dan Suciu -- CSEP544 Fall 2010
B+ Trees Basics

- Parameter $d =$ the \textbf{degree}
- Each node has $\geq d$ and $\leq 2d$ keys (except root)

\begin{center}
\begin{tabular}{c|c|c|c}
 & 30 & 120 & 240 \\
\hline
\end{tabular}
\end{center}

Keys $k < 30$

<table>
<thead>
<tr>
<th>Keys $30 \leq k &lt; 120$</th>
<th>Keys $120 \leq k &lt; 240$</th>
<th>Keys $240 \leq k$</th>
</tr>
</thead>
</table>

- Each leaf has $\geq d$ and $\leq 2d$ keys:

\begin{center}
\begin{tabular}{c|c|c|c}
 & 40 & 50 & 60 \\
\hline
\end{tabular}
\end{center}

<table>
<thead>
<tr>
<th>40</th>
<th>50</th>
<th>60</th>
</tr>
</thead>
</table>

Next leaf
B+ Tree Example

\( d = 2 \)

Find the key 40
B+ Tree Example

d = 2

Find the key 40
B+ Tree Example

d = 2

Find the key 40
B+ Tree Example

d = 2

Find the key 40
Using a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf

- Range queries:
  - As above
  - Then sequential traversal

Index on People(age)

Select name
From People
Where age = 25

Select name
From People
Where 20 <= age and age <= 30
Which queries can use this index?

Index on People(name, zipcode)

Select *
From People
Where name = ‘Smith’
   and zipcode = 12345

Select *
From People
Where name = ‘Smith’

Select *
From People
Where zipcode = 12345
B+ Tree Design

• How large d?
• Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes
• \(2d \times 4 + (2d+1) \times 8 \leq 4096\)
• \(d = 170\)
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%
  - average fanout = 133
- Typical capacities
  - Height 4: $133^4 = 312,900,700$ records
  - Height 3: $133^3 = 2,352,637$ records
- Can often hold top levels in buffer pool
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 Mbytes
Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:
  - If leaf, keep K3 too in right node
  - When root splits, new root has 1 key only
Insertion in a B+ Tree

Insert $K=19$
Insertion in a B+ Tree

After insertion
Now insert 25

Insertion in a B+ Tree
Insertion in a B+ Tree

After insertion
Insertion in a B+ Tree

But now have to split!

Diagram showing the insertion process and the splitting of nodes.
Insertion in a B+ Tree

After the split
Deletion from a B+ Tree

Delete 30
Deletion from a B+ Tree

After deleting 30

May change to 40, or not

Diagram showing the deletion process from a B+ tree.
Deletion from a B+ Tree

Now delete 25
Deletion from a B+ Tree

After deleting 25
Need to rebalance

*Rotate*
Deletion from a B+ Tree

Now delete 40
Delete from a B+ Tree

After deleting 40
Rotation not possible
Need to *merge* nodes
Deletion from a B+ Tree

Final tree
Practical Aspects of B+ Trees

Key compression:

- Each node keeps only the from parent keys
- Jonathan, John, Johnsen, Johnson ...
  - Parent: Jo
  - Child: nathan, hn, hnsen, hnson, ...

Dan Suciu -- CSEP544 Fall 2010
Practical Aspects of B+ Trees

Bulk insertion

- When a new index is created there are two options:
  - Start from empty tree, insert each key one-by-one
  - Do *bulk insertion* – what does that mean?
Practical Aspects of B+ Trees

Concurrency control

- The root of the tree is a “hot spot”
  - Leads to lock contention during insert/delete
- Solution: do proactive split during insert, or proactive merge during delete
  - Insert/delete now require only one traversal, from the root to a leaf
  - Use the “tree locking” protocol
Summary on B+ Trees

- Default index structure on most DBMS
- Very effective at answering ‘point’ queries:
  \[ \text{productName} = \text{‘gizmo’} \]
- Effective for range queries:
  \[ 50 < \text{price} \text{ AND } \text{price} < 100 \]
- Less effective for multirange:
  \[ 50 < \text{price} < 100 \text{ AND } 2 < \text{quant} < 20 \]
Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are n buckets
  - A hash function $f(k)$ maps a key $k$ to $\{0, 1, \ldots, n-1\}$
  - Store in bucket $f(k)$ a pointer to record with key $k$
- Secondary storage: bucket = block, use overflow blocks when needed
Hash Table Example

- Assume 1 bucket (block) stores 2 keys + pointers
- $h(e)=0$
- $h(b)=h(f)=1$
- $h(g)=2$
- $h(a)=h(c)=3$

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>f</td>
</tr>
<tr>
<td>2</td>
<td>g</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>
Searching in a Hash Table

- Search for a:
- Compute $h(a)=3$
- Read bucket 3
- 1 disk access

```
<table>
<thead>
<tr>
<th>0</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>f</td>
</tr>
<tr>
<td>2</td>
<td>g</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>
```
Insertion in Hash Table

- Place in right bucket, if space
- E.g. \( h(d)=2 \)

<table>
<thead>
<tr>
<th>0</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>g</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
</tbody>
</table>
Insertion in Hash Table

- Create overflow block, if no space
- E.g. \( h(k) = 1 \)

More overflow blocks may be needed
Hash Table Performance

- Excellent, if no overflow blocks
- Degrades considerably when number of keys exceeds the number of buckets (i.e. many overflow blocks).
Extensible Hash Table

- Allows hash table to grow, to avoid performance degradation
- Assume a hash function $h$ that returns numbers in \{0, ..., $2k - 1$\}
- Start with $n = 2^i \ll 2^k$, only look at $i$ least significant bits
Extensible Hash Table

- E.g. $i=1$, $n=2^i=2$, $k=4$

- Keys:
  - $4 (=0100)$
  - $7 (=0111)$

- Note: we only look at the last bit (0 or 1)
Insertion in Extensible Hash Table

- Insert 13 (=1101)
Insertion in Extensible Hash Table

- Now insert 0101

- Need to extend table, split blocks
- i becomes 2
Insertion in Extensible Hash Table

- $i=1$
  - (010)0
  - (011)1
  - (110)1, (010)1

- $i=2$
  - (010)0
  - (11)01
  - (01)01
  - (01)11

- (01)01
Insertion in Extensible Hash Table

- Now insert 0000, 1110

- Need to split block
Insertion in Extensible Hash Table

- After splitting the block
Extensible Hash Table

- How many buckets (blocks) do we need to touch after an insertion?

- How many entries in the hash table do we need to touch after an insertion?
Performance Extensible Hash Table

- No overflow blocks: access always one read
- BUT:
  - Extensions can be costly and disruptive
  - After an extension table may no longer fit in memory
Linear Hash Table

- Idea: extend only one entry at a time
- Problem: \( n \) is no longer a power of 2
- Let \( i \) be such that \( 2^i \leq n < 2^{i+1} \)
- After computing \( h(k) \), use last \( i \) bits:
  - If last \( i \) bits represent a number > \( n \), change msb from 1 to 0 (get a number \( \leq n \))
Linear Hash Table Example

- \( n=3 \)
Linear Hash Table Example

• Insert 1000: overflow blocks…
Linear Hash Tables

- Extension: independent on overflow blocks
- Extend $n := n + 1$ when average number of records per block exceeds (say) 80%
Linear Hash Table Extension

- From $n=3$ to $n=4$

Only need to touch one block (which one?)

- $n=11$
Linear Hash Table Extension

- From $n=3$ to $n=4$ finished

- Extension from $n=4$ to $n=5$ (new bit)

- Need to touch every single block (why?)
Indexes in Postgres

CREATE TABLE V(M int, N varchar(20), P int);

CREATE INDEX V1_N ON V(N)

CREATE INDEX V2 ON V(P, M)

CREATE INDEX VVV ON V(M, N)

CLUSTER V USING V2

Makes V2 clustered
Database Tuning Overview

• The database tuning problem
• Index selection (discuss in detail)
• Horizontal/vertical partitioning (see lecture 3)
• Denormalization (discuss briefly)
Levels of Abstraction in a DBMS

- **Physical Schema**: includes storage details, file organization, indexes.
- **Conceptual Schema**: a.k.a logical schema, describes stored data in terms of data model.
- **External Schema**: views, access control.

Disk
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: 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
Index Selection Problem 1

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

Your workload is this

100000 queries:

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

100 queries:

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

What indexes?
Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

SELECT *
FROM V
WHERE N=?

100 queries:

SELECT *
FROM V
WHERE P=?

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

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

Your workload is this:

100,000 queries:

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

100 queries:

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

100,000 queries:

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

What indexes?
Index Selection Problem 2

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

Your workload is this:

- 100000 queries:
  - `SELECT * FROM V WHERE N>? and N<?`
- 100 queries:
  - `SELECT * FROM V WHERE P=?`
- 100000 queries:
  - `INSERT INTO V VALUES (?, ?, ?)`

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

V(M, N, P);

Your workload is this

100000 queries:

SELECT *
FROM V
WHERE N=?

1000000 queries:

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

1000000 queries:

INSERT INTO V
VALUES (?, ?, ?)

100000 queries:

What indexes?
Index Selection Problem 3

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

Your workload is this

100000 queries:  SELECT * FROM V WHERE N=?

100000 queries:  SELECT * FROM V WHERE N=? and P>?

100000 queries:  INSERT INTO V VALUES (?, ?, ?)

A:  \[ V(N, P) \]
Index Selection Problem 4

V(M, N, P);

Your workload is this

1000 queries:
SELECT * FROM V WHERE N>? and N<?

100000 queries:
SELECT * FROM V WHERE P>? and P<?

What indexes?
Index Selection Problem 4

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

Your workload is this

1000 queries:

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

100000 queries:

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

A: \( V(N) \) secondary, \( V(P) \) primary index
The Index Selection Problem

• **SQL Server**
  - Automatically, thanks to *AutoAdmin* project
  - Much acclaimed successful research project from mid 90’s, similar ideas adopted by the other major vendors

• **PostgreSQL**
  - You will do it manually, part of homework 5
  - But tuning wizards also exist
Index Selection: Multi-attribute Keys

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

```
SELECT K2 FROM R WHERE K1=55
```
To Cluster or Not

- Range queries benefit mostly from clustering
- Covering indexes do not need to be clustered: they work equally well unclustered
SELECT * FROM R WHERE K>? and K<?
Hash Table v.s. B+ tree

- 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 (you will understand that in a few lectures)
Balance Queries v.s. Updates

- Indexes speed up queries
  - SELECT FROM WHERE
- But they usually slow down updates:
  - INSERT, DELECTE, UPDATE
  - However some updates benefit from indexes
    
    \[
    \text{UPDATE R} \\
    \text{SET A = 7} \\
    \text{WHERE K=55}
    \]
Tools for Index Selection

• SQL Server 2000 Index Tuning Wizard
• DB2 Index Advisor

• How they work:
  - They walk through a large number of configurations, compute their costs, and choose the configuration with minimum cost
Tuning the Conceptual Schema

- Denormalization
- Horizontal Partitioning
- Vertical Partitioning
Denormalization

Product(\textbf{pid}, \texttt{pname}, \texttt{price}, \texttt{cid})
Company(\textbf{cid}, \texttt{cname}, \texttt{city})

A very frequent query:

\begin{verbatim}
SELECT x.pid, x.pname
FROM Product x, Company y
WHERE x.cid = y.cid and x.price < ? and y.city = ?
\end{verbatim}

How can we speed up this query workload? 2010
Denormalization

Product\((pid, \text{pname}, \text{price}, \text{cid})\)
Company\((cid, \text{cname}, \text{city})\)

Denormalize:
ProductCompany\((pid, \text{pname}, \text{price}, \text{cname}, \text{city})\)

```
INSERT INTO ProductCompany
SELECT x.pid, x.pname, price, y.cname, y.city
FROM Product x, Company y
WHERE x.cid = y.cid
```
Denormalization

Next, replace the query

```sql
SELECT x.pid, x.pname
FROM Product x, Company y
WHERE x.cid = y.cid and x.price < ? and y.city = ?
```

```sql
SELECT pid, pname
FROM ProductCompany
WHERE price < ? and city = ?
```
Issues with Denormalization

- It is no longer in BCNF
  - We have the hidden FD: cid → cname, city

- When Product or Company are updated, we need to propagate updates to ProductCompany
  - Use RULE in postgres (see below)
  - Or use a trigger on a different RDBMS

- Sometimes cannot modify the query
  - What do we do then?
Denormalization Using Views

\[
\text{INSERT INTO} \ \text{ProductCompany} \\
\quad \text{SELECT} \ x.pid, x.pname, .price, y.cid, y.cname, y.city \\
\quad \text{FROM} \ \text{Product} \ x, \ \text{Company} \ y \\
\quad \text{WHERE} \ x.cid = y.cid; \\
\]

\[
\text{DROP Product; DROP Company;} \\
\]

\[
\text{CREATE VIEW Product AS} \\
\quad \text{SELECT} \ \text{pid, pname, price, cid FROM ProductCompany} \\
\]

\[
\text{CREATE VIEW Company AS} \\
\quad \text{SELECT DISTINCT cid, cname, city FROM ProductCompany} \\
\]
Denormalization Using Views

Keep the query unchanged

```
SELECT x.pid, x.pname
FROM Product x, Company y
WHERE x.cid = y.cid and x.price < ? and y.city = ?
```

What does the system do?
Denormalization Using Views

- In postgres the rewritten query is non-minimal:
  - Means: has redundant joins
  - To see this in postgres, type “explain . . .”
  - For Project 2: it’s OK to use denormalization using views (don’t forget indexes); performance is reasonable

- SQL Server does a better job with this query
Horizontal Partition

Product($\text{pid}$, $\text{pname}$, $\text{price}$, $\text{cid}$)

Horizontal partition on price $< 10$ and price $\geq 10$

- When few products have price $< 10$ but most queries are about these products
Horizontal Partition

\[
\text{INSERT INTO} \quad \text{CheapProduct} \quad \ldots \quad \text{WHERE} \quad \text{price} < 10 \\
\text{INSERT INTO} \quad \text{ExpensiveProduct} \quad \ldots \quad \text{WHERE} \quad \text{price} \geq 10
\]

\text{DROP Product}

\text{CREATE VIEW} \quad \text{Product} \quad \text{AS} \\
(\text{select} \quad \ast \quad \text{from} \quad \text{cheapProduct}) \quad \text{UNION ALL} \\
(\text{select} \quad \ast \quad \text{from} \quad \text{expensiveProduct})
Horizontal Partition

\[
\begin{align*}
&\text{SELECT } * \\
&\text{FROM } \text{Product} \\
&\text{WHERE } \text{price} = 2
\end{align*}
\]

Which of the tables \text{cheapProduct} and \text{expensiveProduct} does it touch?
Horizontal Partition

- The query will touch both cheapProduct and expensiveProduct because we haven’t told the system the partition criteria (price < 10 and >= 10)
- We can do this in two ways:
  - As a predicate in the view definition
  - As a constraint in the table definition
Partition Criteria As View Predicates

```sql
CREATE VIEW Product AS
 (select * from cheapProduct where price < 10)
 UNION ALL
 (select * from expensiveProduct where price >= 10)
```

SQL Server correctly optimizes the query, but postgres doesn’t
Partition Criteria As Table Constraints

CREATE TABLE CheapProduct (  
    pid int primary key not null,  
    pname varchar(20) not null,  
    price int not null,  
    CHECK (price < 10));  

CREATE TABLE ExpensiveProduct (  
    . . .  
    CHECK (price >= 10));

If you set “constraint_exclusion = on” in postgresql.conf, then postgres optimizes this line.
Updates Through Views

- Product is a view:
  - What should “INSERT INTO Product” do?

- Sometime it is possible for the system to figure out which base tables to update

- If not, then use RULES or TRIGGERS
RULES in Postgres

CREATE [ OR REPLACE ] RULE name AS ON event TO table [ WHERE condition ]
DO [ ALSO | INSTEAD ] { NOTHING |
command | ( command ; command ... ) }

Where
name = a name for the rule
event = SELECT, INSERT, UPDATE, or DELETE
command = SELECT, INSERT, UPDATE, DELETE
use new for the new tuple, and old for the old tuple
RULES in Postgres

CREATE OR REPLACE RULE productInsertRule AS
ON INSERT TO Product DO INSTEAD
  (INSERT INTO cheapProducts
    SELECT DISTINCT new.pid, new.pname, new.price
    FROM anyDummyTablePreferablyWithOneTuple
    WHERE new.price < 10;
  INSERT INTO expensiveProducts
    SELECT DISTINCT new.pid, new.pname, new.price
    FROM anyDummyTablePreferablyWithOneTuple
    WHERE new.price >= 10);
CREATE OR REPLACE RULE productDeleteRule AS
ON DELETE TO Product DO INSTEAD
  (DELETE FROM cheapProducts
   WHERE pid = old.pid
  DELETE FROM expensiveProducts
   WHERE pid = old.pid);
Vertical Partition

Product(\textbf{pid}, \textit{pname}, \textit{price}, \textit{description})

Split vertically into:
   Product1(\textit{pid}, \textit{name}, \textit{price})
   Product2(\textit{pid}, \textit{description})

Define Product as view
CREATE VIEW Product AS
(select x.pid, x.pname, x.price, y.description
from Product1 x, Product2 y
where x.pid = y.pid)
Vertical Partition

Now consider a query on Product:

```
SELECT pid, pname
FROM Product
WHERE price > 20
```

Which tables are touched by the system?
Vertical Partition

• SQL Server does the right thing:
  - Touches only product1
• But postgres insists on joining product1 with product2 instead
  - I couldn’t figure out how to coerce postgres to optimize this query
  - 10 bonus points for whoever finds out first!
  - In the meantime, we will cheat like this:
CREATE VIEW Product AS
select pid, pname, price, 'blah' as description
from Product1
NOT DISCUSSED IN CLASS
Security in SQL

- Discretionary access control in SQL
- Using views for security
Discretionary Access Control in SQL

GRANT privileges
ON object
TO users
[WITH GRANT OPTIONS]

privileges = SELECT | INSERT(column-name) | UPDATE(column-name) | DELETE | REFERENCES(column-name)

object = table | attribute
Examples

GRANT INSERT, DELETE ON Customers TO **Yuppy** WITH GRANT OPTIONS

Queries allowed to Yuppy:

- INSERT INTO Customers(cid, name, address) VALUES(32940, ‘Joe Blow’, ‘Seattle’)
- DELETE Customers WHERE LastPurchaseDate < 1995

Queries denied to Yuppy:

- SELECT Customer.address FROM Customer WHERE name = ‘Joe Blow’
Examples

GRANT SELECT ON Customers TO Michael

Now Michael can SELECT, but not INSERT or DELETE
Examples

GRANT SELECT ON Customers
    TO Michael WITH GRANT OPTIONS

Michael can say this:
    GRANT SELECT ON Customers TO Yuppi

Now Yuppi can SELECT on Customers
Examples

GRANT UPDATE (price) ON Product TO Leah

Leah can update, but only Product.price, but not Product.name
Examples

Customer(cid, name, address, balance)
Orders(oid, cid, amount)    cid= foreign key

Bill has INSERT/UPDATE rights to Orders.
BUT HE CAN’T INSERT ! (why ?)

GRANT REFERENCES (cid) ON Customer TO Bill

Now Bill can INSERT tuples into Orders
**Views and Security**

**Customers:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

CREATE VIEW PublicCustomers
SELECT Name, Address
FROM Customers
GRANT SELECT ON PublicCustomers TO Fred

David owns

Fred is not allowed to see this

David says
Views and Security

Customers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

CREATE VIEW BadCreditCustomers
  SELECT *
  FROM Customers
  WHERE Balance < 0
GRANT SELECT ON BadCreditCustomers TO John

David says

John is allowed to see only <0 balances.
Views and Security

- Each customer should see only her/his record

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

CREATE VIEW CustomerMary
    SELECT * FROM Customers
    WHERE name = 'Mary'
    GRANT SELECT ON CustomerMary TO Mary

CREATE VIEW CustomerSue
    SELECT * FROM Customers
    WHERE name = 'Sue'
    GRANT SELECT ON CustomerSue TO Sue

Doesn’t scale.
Need *row-level* access control!
Revocation

REVOKE [GRANT OPTION FOR] privileges
    ON object FROM users { RESTRICT | CASCADE }

Administrator says:

REVOKE SELECT ON Customers FROM David CASCADE

John loses SELECT privileges on BadCreditCustomers
Revocation

Joe: GRANT [....] TO Art ...
Art: GRANT [....] TO Bob ...
Bob: GRANT [....] TO Art ...
Joe: GRANT [....] TO Cal ...
Cal: GRANT [....] TO Bob ...
Joe: REVOKE [....] FROM Art CASCADE

What happens ??
Revocation

According to SQL everyone keeps the privilege
Summary of SQL Security

Limitations:

• No row level access control
• Table creator owns the data: that’s unfair!
• Today the database is not at the center of the policy administration universe