Overview of Relational DBMS (CS 4320 Recap)

CS 6320
Overview

Architecture of database systems, Hellerstein et al., 2007.
Overview

Architecture of database systems, Hellerstein et al., 2007.
Creating Relations in SQL

- Creates Students relation
  - Type (domain) of each field is specified
  - Enforced by DBMS whenever tuples are added or modified
- Enrolled table holds information about courses that students take

CREATE TABLE Students
(sid CHAR(20),
name CHAR(20),
login CHAR(10),
age INT,
gpa REAL);

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2));
Foreign Keys in SQL

Only students listed in the Students relation should be allowed to enroll for courses

```sql
CREATE TABLE Enrolled
    (sid CHAR(20), cid CHAR(20), grade CHAR(2),
     PRIMARY KEY (sid,cid),
     FOREIGN KEY (sid) REFERENCES Students (sid) );
```

<table>
<thead>
<tr>
<th>Enrolled</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>cid</td>
<td>grade</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53666</td>
<td>Reggae203</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Students</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
<td>login</td>
<td>age</td>
<td>gpa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inserting Data

INSERT INTO Students
VALUES ('5', 'Thomas', 'Th75', 20, 3.7);
SELECT  S.name, E.cid
FROM     Students S, Enrolled E
WHERE  S.sid=E.sid AND S.gpa>3.5;
SQL Summary:

- Basic SELECT/FROM/WHERE queries
- Expressions and strings
- Set operators
- Nested queries
- Aggregation
- GROUP BY/HAVING
- Null values and Outer Joins
- (ORDER BY and other features…)
Overview

Architecture of database systems, Hellerstein et al., 2007.
Query Optimization Overview

Query Parser

Query Optimizer

- Plan generator
- Plan cost estimator

Catalog Manager

Physical Query Plan

Query Plan Evaluator
Optimization

Query: $R \Join S \Join T$
Optimization

Query: R⨝S⨝T

Optimal Plan

Sub-Optimal Plans

Time
Optimization

Query: $R \bowtie S \bowtie T$
Optimization

Query: R $\Join$ S $\Join$ T

- R $\Join$ S
- S $\Join$ T
- R $\Join$ T
- R
- S
- T
Optimization

Query: $R \Join S \Join T$
Optimization

Query: $R \bowtie S \bowtie T$
Optimization

Query: \( R \bowtie S \bowtie T \)
Optimization

Query: $R \bowtie S \bowtie T$
Enumeration of Plans

- Pass 1: Find best 1-relation plan for each relation
  - includes any selects/projects just on this relation.
- Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. (*All 2-relation plans.*)
- Pass k: Find best way to join result of a (k-1)-relation plan (as outer) to the kth relation. (*All k-relation plans.*)
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND R.bid=100 AND S.rating>5

Logical query plan:

Physical query plan = RA tree annotated with info on access methods and operator implementation
**Tuple Nested Loop Join**

```plaintext
foreach tuple r in R do
    foreach tuple s in S do
        if r.sid == s.sid then add <r, s> to result
```

- R is “outer” relation
- S is “inner” relation
Page Nested Loop Join

foreach page p1 in R do
  foreach page p2 in S do
    foreach r in p1 do
      foreach s in p2 do
        if r.sid == s.sid then add <r, s> to result

R is "outer" relation
S is "inner" relation
Block Nested Loops Join

Use one page as input buffer for scanning $S$, one page as output buffer, and all remaining pages to hold \textquoteleft\textquoteleft block\textquoteright\ of $R$.

- For each matching tuple $r$ in R-block, $s$ in $S$-page, add $<r, s>$ to result. Then read next R-block, scan $S$, etc.
Index Nested Loops Join

foreach tuple r in R do
    foreach tuple s in S where $r_i == s_j$ do
        add <r, s> to result

Suppose we have an index on S, on the join attribute

No need to scan all of S – just use index to retrieve tuples that match this r

This will probably be faster, especially if there are few matching tuples and the index is clustered
Sort-Merge Join

Sort R and S on the join column, then scan them to do a "merge" (on join col.), and output result tuples.
Partition both relations using hash fn \( h \): R tuples in partition i will only match S tuples in partition i.
Overview

Architecture of database systems, Hellerstein et al., 2007.
Tree-structured indexing

Tree-structured indexing techniques support both *range searches* and *equality searches*.

- **ISAM**: static structure; **B+ tree**: dynamic, adjusts gracefully under inserts and deletes.

Simple cost metric for discussion of search costs: number of disk I/Os (i.e. how many pages need to be brought in from disk)

- Ignore benefits of sequential access etc to simplify
B+ Tree Indexes

- Leaf pages contain *data entries*
- Non-leaf pages have *index entries*; only used to direct searches:
Clustered vs. Unclustered Index

**Clustered Index:**
- Data entries directly connected to index entries.
- Direct search for data entries.
- Index entries are clustered together.

**Unclustered Index:**
- Data entries not directly connected to index entries.
- Need to search index entries to find data entries.
- Index entries are not clustered together.

Data Records vs. Data File:
- Data Records are connected to data entries.
- Data File is separate from data entries.

Diagram illustrates the difference in organization and search methods between clustered and unclustered indexes.
Indexing using Hashing

Hash-based indexes are for equality selections. Cannot support range searches.

Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.
Overview

Architecture of database systems, Hellerstein et al., 2007.
Buffer Management in a DBMS

Page Requests from Higher Levels

MAIN MEMORY

BUFFER POOL

DISK

DB

choice of frame dictated by replacement policy

Data must be in RAM for DBMS to operate on it!
Table of <frame#, pageid> pairs is maintained.
When a Page is Requested ...

- If page is not in pool (cache miss):
  - Choose a frame for replacement
  - If frame contains a page with changes, write it to disk
  - Read requested page into chosen frame
  - Pin the page and return its address.
- If requested page is in pool (cache hit):
  - Increment its pin count and return its address.

If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time.
Buffer Replacement Policies

- Lots of other replacement policies:
  - MRU
  - LFU (Least Frequently Used)
  - Random
  - FIFO (First In First Out)
  - Clock (Round Robin)

- Different benefits for different workloads
  - Also, some require keeping less state than others
Policy can have big impact on # of I/O’s; depends on the access pattern.

Sequential flooding: Nasty situation caused by LRU + repeated sequential scans.
- # buffer frames < # pages in file means each page request causes an I/O.
- Example scenario: join implementation with nested loops
Overview

Architecture of database systems, Hellerstein et al., 2007.
Transactions

- Are a fundamental database abstraction
- ACID properties
  - Atomicity
  - Durability
  - Consistency
  - Isolation
- Broadly supported in relational DBMSs
- NoSQL support is a moving target
Atomicity

- A transaction should execute completely or not at all.
- If the first few statements succeed, but the next one fails, the entire transaction must be rolled back.
  - This failure could be due to an error/exception or to a system crash.
- It ain't over till it's over – nothing is guaranteed until the transaction commits.
Assume we have an intrinsic notion of data consistency
  - E.g. semantic constraints are satisfied by DB
    - E.g. every order has associated billing info

The "C" in ACID: A transaction, if executed by itself on a consistent DB, will produce another consistent DB
  - An assumption that a transaction is a self-contained unit of work (no loose ends)
Isolation

- No harmful interference between transactions is permitted as they run
- Every transaction should have the illusion of having the DB to itself
Once a transaction does **commit**, the changes should be **persistent**

If system crashes before changes make it to disk, this could be a problem!

Does not preclude the ability to "undo" a real world action, e.g. cancel an order
  - But this must be done using a second transaction.
Overview

Architecture of database systems, Hellerstein et al., 2007.
Big Picture (all inclusions are proper)

All schedules

Final State

Serializable

View Serializable

Conflict

Serializable

Serial
Conflict Graphs

Given a schedule, can identify all conflicting pairs of operations and represent them as a graph.

- Nodes are transactions.
- Edge from i to j if transaction i contains an operation that conflicts with and precedes (in the schedule) an operation by transaction j.

Example: R1(A) W2(A) R1(A)
A schedule is conflict serializable if its conflict graph contains no cycle

Alternative (equivalent) statement: it is conflict serializable if it has the same conflict graph as some serial schedule

- Why are these equivalent?

Topological sort on the conflict graph gives us equivalent serial execution
First family of protocols – based on idea of locks

Before any read or write, a transaction must request a lock on an object
– A "permission to operate" on this object

Locks are managed centrally by the DBMS lock manager
## 2PL variants

<table>
<thead>
<tr>
<th>2PL variants</th>
<th>Conservative</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td><img src="image1" alt="Graph" /></td>
<td><img src="image2" alt="Graph" /></td>
</tr>
<tr>
<td>No</td>
<td><img src="image3" alt="Graph" /></td>
<td><img src="image4" alt="Graph" /></td>
</tr>
</tbody>
</table>
Optimistic CC

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - These overheads occur even if conflicts are rare.

- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before commit.
System keeps several versions of each data item
When a transaction writes a data item, it creates a new version rather than overwriting
When a transaction reads a data item, the version visible to the read is determined by the protocol used (several options)
Maintaining versions can be nontrivial and comes with its own extra cost, of course
Overview

Architecture of database systems, Hellerstein et al., 2007.
Basic Idea: Logging

• Record REDO and UNDO information, for every update, in a log that will survive crashes.
  - Log is written sequentially.
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.

• Log: An ordered list of REDO/UNDO actions
  - Log record contains:
    <transID, pageID, offset, length, old data, new data>
  - and additional control info (which we’ll see soon).
Write-Ahead Logging (WAL)

The Write-Ahead Logging Protocol:
- Must force the log record for an update before the corresponding data page gets to disk.
- Must write all log records for a transaction before commit.

#1 guarantees Atomicity (why?)
#2 guarantees Durability (why?)
Exactly how is logging (and recovery!) done?
- We’ll study the ARIES algorithm.
Crash Recovery: Big Picture

Start from a **checkpoint** (found via **master** record).

Three phases. Need to:

- Figure out which transactions committed since checkpoint, which failed (**Analysis**).
  - **REDO** all actions.
  - **UNDO** effects of failed transactions.
Overview