CMSC 424 – Database design
Lecture 20
Join size prediction
Concurrency/recovery

Mihai Pop

Admin

Job Description

Software Developer

This Software Developer position is located at the U.S. Army Medical Research and Materiel Command (MRMC) Bioinformatics Cell (BIC), Ft. Detrick, MD. Applicants should have a Bachelors or Masters degree in computer science, engineering, or a related discipline and an interest in pursuing the development of software tools that directly support the mission of the MRMC life scientists. In particular, the Software Developer supports biomedical research by transitioning proof of concept software prototypes to production quality systems for research applications. This position requires substantial experience developing Web enabled client/server systems with a database backend. A strong knowledge of servelets, CGI, HTML, JavaScript, SQL, XML, Java applets, and GNU tools is desirable. Must know C and possess good knowledge of OO, MySQL/Oracle, and Python/Ruby/LISP. Operating system knowledge should include Unix variants (BSD's, GNU/Linux) and Windows OS. The Software Developer will also help maintain the BIC's IT infrastructure. Experience with installing and maintaining UNIX/LINUX based server machines, and a working knowledge of Local Area Networks, MS Exchange based mail servers, DNS servers and CISCO firewalls is a plus.

Please submit resume to:

Jaques Reifman, Ph.D.
Senior Research Scientist
U.S. Army Medical Research and Material Command
Ft. Detrick, MD
Email: reifman@bioanalysis.org
Phone: 301-619-7915
www.bioanalysis.org

Statistical Information for Cost Estimation

- n_r: number of tuples in a relation *r*.
- b_r: number of blocks containing tuples of *r*.
- l_r: size of a tuple of *r*.
- f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
- *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\Pi_A(r)$.
- If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

- Also: instead of V(A,r) can store a histogram

Selection Size Estimation

- $\sigma_{A=v}(r)$
  - n_r / $V(A,r)$: number of records that will satisfy the selection
  - Equality condition on a key attribute: *size estimate* = 1
- $\sigma_{A \leq V}(r)$ (case of $\sigma_{A \geq V}(r)$ is symmetric)
  - Let c denote the estimated number of tuples satisfying the condition.
  - If min(A,r) and max(A,r) are available in catalog
    - c = 0 if v < min(A,r)
    - c = $n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information c is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

- The **selectivity** of a condition θ_i is the probability that a tuple in the relation *r* satisfies θ_i .
  - If s_i is the number of satisfying tuples in *r*, the selectivity of θ_i is given by s_i / n_r.
- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. *Assuming indepdence,* estimate of tuples in the result is: $n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$
- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Estimated number of tuples:

$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$

- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples:

$$n_r - size(\sigma_\theta(r))$$

Join Operation: Running Example

Running example:

$depositor \bowtie customer$

Catalog information for join examples:

- $n_{customer} = 10{,}000.$
- $f_{customer} = 25$, which implies that
  $b_{customer} = 10000/25 = 400.$
- $n_{depositor} = 5000.$
- $f_{depositor} = 50$, which implies that
  $b_{depositor} = 5000/50 = 100.$
- *V(customer_name, depositor)* = 2500, which implies that , on average, each customer has two accounts.
  - Also assume that *customer_name* in *depositor* is a foreign key on *customer.*
  - *V(customer_name, customer) = 10000 (primary key!)*

Estimation of the Size of Joins

- The Cartesian product r x s contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
- If $R \cap S = \emptyset$, then $r \bowtie s$ is the same as r x s.
- If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s.
- If $R \cap S$ *in* S is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
    - The case for $R \cap S$ being a foreign key referencing S is symmetric.
- In the example query *depositor* $\bowtie$ *customer, customer_name* in *depositor* is a foreign key of *customer*
  - hence, the result has exactly $n_{depositor}$ tuples, which is 5000

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that every tuple t in R produces tuples in $R \bowtie S$, the number of tuples in $R \bowtie S$ is estimated to be:

$$\frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

$$\frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.
- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

Estimation of the Size of Joins (Cont.)

- Compute the size estimates for *depositor* ⋈ *customer* without using information about foreign keys:
  - *V(customer_name, depositor)* = 2500, and *V(customer_name, customer)* = 10000
  - The two estimates are 5000 * 10000/2500 - 20,000 and 5000 * 10000/10000 = 5000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\prod_A(r)$ = $V(A,r)$
- Aggregation : estimated size of ${}_A\boldsymbol{g}_F(r)$ = $V(A,r)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g. $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1}\sigma_{\theta 2}(r)$
  - For operations on different relations:
    - estimated size of $r \cup s$ = size of r + size of s.
    - estimated size of $r \cap s$ = minimum size of r and size of s.
    - estimated size of $r - s$ = r.
    - <u>All the three estimates may be quite inaccurate, but provide upper bounds on the sizes</u>.

Transactions, concurrency, recovery

- Until now we learned
  - how to design an efficient database
  - how to quickly answer queries

- Next: how we ensure database is consistent:
  - equipment failures (disk, power, internet...)
  - concurrent accesses

Overview

- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - ***Atomicity***: Entire transaction or nothing
  - ***Consistency***: Transaction, executed completely, takes database from one consistent state to another
  - ***Isolation***: Concurrent transactions *appear* to run in isolation
  - ***Durability***: Effects of committed transactions are not lost
- Consistency: Transaction programmer needs to guarantee that
    - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees

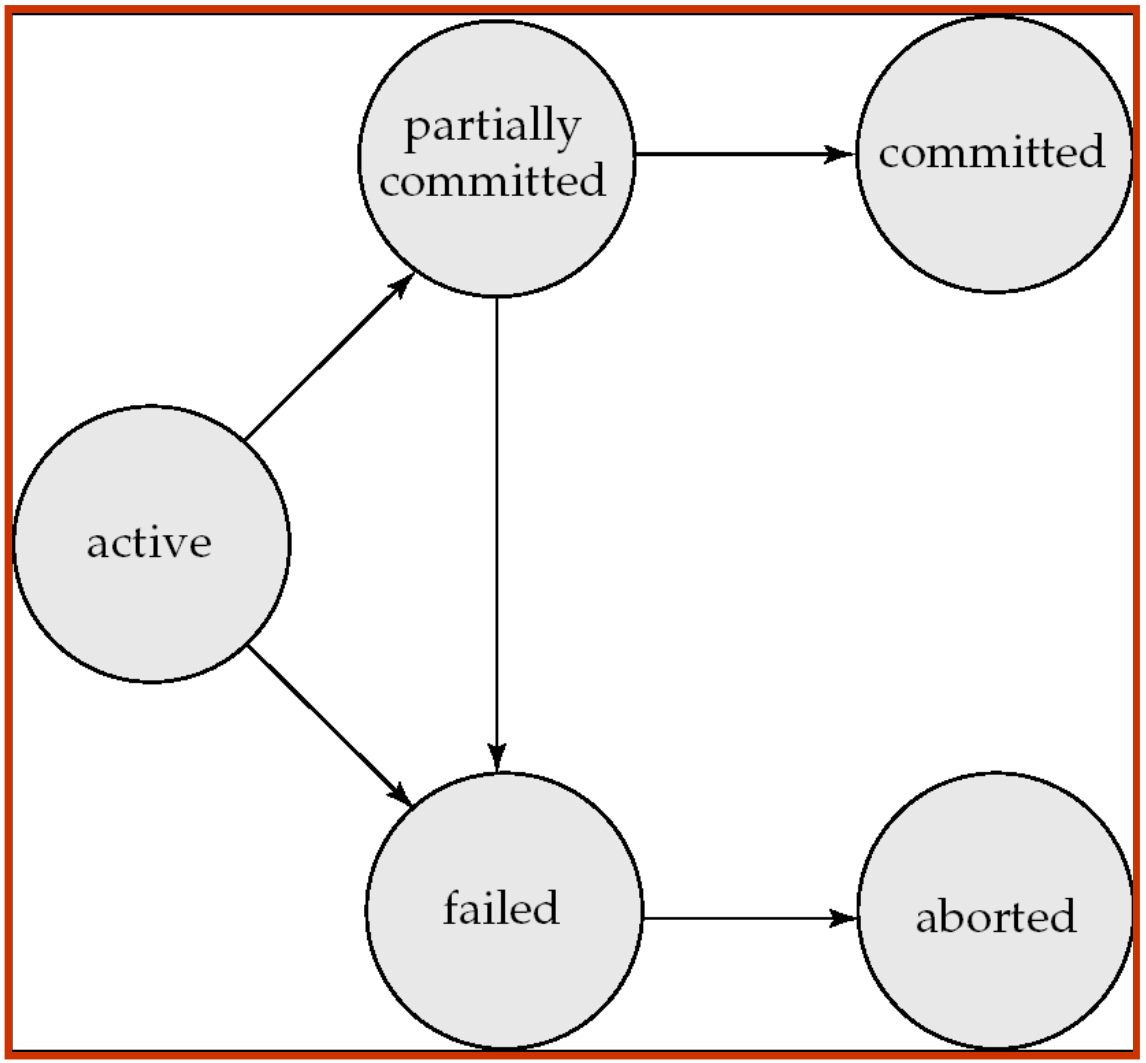
Assumptions and Goals

- Assumptions:
  - The system can crash at any time
  - Similarly, the power can go out at any point
    - Contents of the main memory won't survive a crash, or power outage
  - BUT… **disks are durable. They might stop, but data is not lost**
  - Disks only guarantee *atomic sector writes*, nothing more
  - Transactions are by themselves consistent
- Goals:
  - Guaranteed durability, atomicity
  - As much concurrency as possible, while not compromising isolation and/or consistency
    - Two transactions updating the same account balance… NO
    - Two transactions updating different account balances… YES

Next…

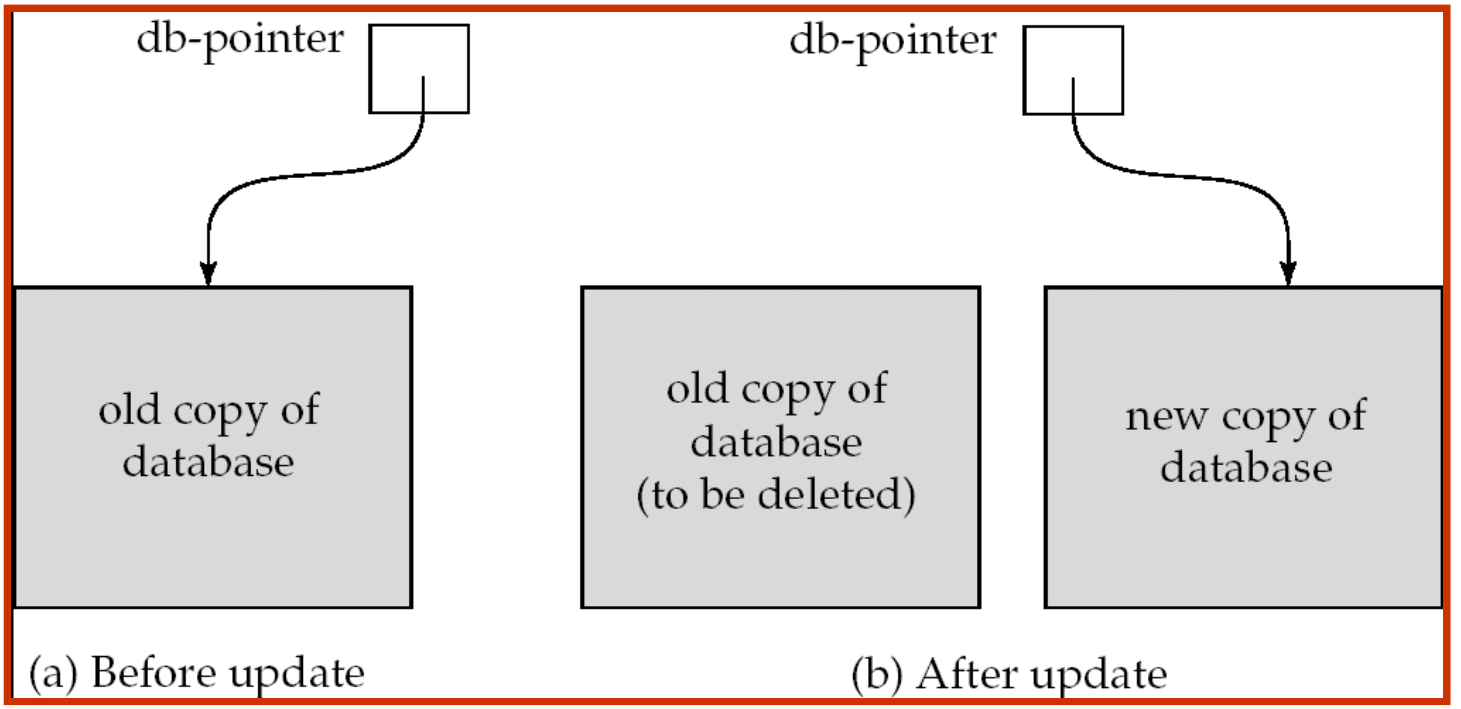
- States of a transaction
- A simple solution called *shadow copy*
  - Satisfies Atomicity, Durability, and Consistency, but no Concurrency
  - Very inefficient

Transaction states



Shadow Copy

- Make updates on a copy of the database.
- Switch pointers atomically after done.
  - Some *text editors* work this way



Shadow Copy

- Atomicity:
  - As long as the DB pointer switch is atomic.
    - Okay if DB pointer is in a single block
- Concurrency:
  - No.
- Isolation:
  - No concurrency, so isolation is guaranteed.
- Durability:
  - Assuming disk is durable (we will assume this for now).
- Very inefficient:
  - Databases tend to be very large. Making extra copies not feasible. Further, no concurrency.

Next…

- Concurrency control schemes
  - A CC scheme is used to guarantee that concurrency does not lead to problems
  - For now, we will assume durability is not a problem
    - So no crashes
    - Though transactions may still abort

- Schedules

- When is concurrency okay ?
  - Serial schedules
  - Serializability

A Schedule

Transactions:

T1: transfers $50 from A to B

T2: transfers 10% of A to B

Database constraint: A + B is constant (*checking+saving accts)*

T1	T2
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

Each transaction obeys the constraint.

This schedule does too.

Schedules

- A *schedule* is simply a (possibly interleaved) execution sequence of transaction instructions

- *Serial Schedule:* A schedule in which transaction appear one after the other
  - ie., No interleaving

- Serial schedules satisfy isolation and consistency
  - Since each transaction by itself does not introduce inconsistency

Example Schedule

- Another "serial" schedule:

T1	T2
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	

Effect:	Before	After
A	100	40
B	50	110

Consistent ?

Constraint is satisfied.

Since each Xion is consistent, any serial schedule must be consistent

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Consistent.
So this schedule is okay too.

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Further, the effect same as the serial schedule 1.

Called *serializable*

Example Schedules (Cont.)

A "bad" schedule

T1	T2
read(A)	
A = A -50	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
write(A)	
read(B)	
B=B+50	
write(B)	
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	50
B	50	60

Not consistent

Serializability

- A schedule is called *serializable* if its final effect is the same as that of a *serial schedule*

- Serializability → schedule is fine and does not result in inconsistent database
  - Since serial schedules are fine

- Non-serializable schedules are unlikely to result in consistent databases

- We will ensure serializability
  - Typically relaxed in real high-throughput environments

Serializability

- Not possible to look at all n! serial schedules to check if the effect is the same
  - Instead we ensure serializability by allowing or not allowing certain schedules
- Conflict serializability
- View serializability
- View serializability allows more schedules