Lecture 27:

Friday, December 6, 2002

Outline

- Cost estimation: 16.4
- Recovery using undo logging 17.2

Estimating Sizes

- Need size in order to estimate cost
- Example:
  - Cost of partitioned hash-join E1 \bowtie E2
    is 3B(E1) + 3B(E2)
  - B(E1) = T(E1) * record size/ block size
  - B(E2) = T(E2) * record size/ block size
  - So, we need to estimate T(E1), T(E2)

Estimating Sizes

Estimating the size of a projection

- Easy: T(\Pi_1(R)) = T(R)
- This is because a projection doesn’t eliminate duplicates

Estimating Sizes

Estimating the size of a natural join, R \bowtie_A S

- When the set of A values are disjoint, then T(R \bowtie_A S) = 0
- When A is a key in S and a foreign key in R, then T(R \bowtie_A S) = T(R)
- When A has a unique value, the same in R and S, then T(R \bowtie_A S) = T(R) T(S)
Estimating Sizes

Assumptions:
- Containment of values: if \( V(R, A) \subseteq V(S, A) \), then the set of \( A \) values of \( R \) is included in the set of \( A \) values of \( S \)
  - Note: this indeed holds when \( A \) is a foreign key in \( R \), and a key in \( S \)
- Preservation of values: for any other attribute \( B \),
  \( V(R \bowtie_A S, B) = V(R, B) \) (or \( V(S, B) \))

Estimating Sizes

Assume \( V(R, A) \subseteq V(S, A) \)
- Then each tuple \( t \) in \( R \) joins some tuple(s) in \( S \)
  - How many?
  - On average \( T(S) \cdot V(S, A) \)
  - \( t \) will contribute \( T(S) / V(S, A) \) tuples in \( R \bowtie_A S \)
- Hence \( T(R \bowtie_A S) = T(R) \cdot T(S) / V(S, A) \)

In general: \( T(R \bowtie_A S) = T(R) \cdot T(S) / \max(V(R, A), V(S, A)) \)

Estimating Sizes

Example:
- \( T(R) = 10000 \), \( T(S) = 20000 \)
- \( V(R, A) = 100 \), \( V(S, A) = 200 \)
- How large is \( R \bowtie_A S \)?

Answer: \( T(R \bowtie_A S) = 10000 \cdot 20000 / 200 = 1 \text{M} \)

Estimating Sizes

Joins on more than one attribute:
- \( T(R \bowtie_{A,B} S) = T(R) \cdot T(S) / \max(V(R, A), V(S, A)) \cdot \max(V(R, B), V(S, B)) \)

Histograms

• Statistics on data maintained by the RDBMS
• Makes size estimation much more accurate (hence, cost estimations are more accurate)

Histograms

Employee\((\text{ssn, name, salary, phone})\)
• Maintain a histogram on salary:

<table>
<thead>
<tr>
<th>Salary</th>
<th>0-20k</th>
<th>20k-40k</th>
<th>40k-60k</th>
<th>60k-80k</th>
<th>80k-100k</th>
<th>&gt; 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuples</td>
<td>200</td>
<td>300</td>
<td>5000</td>
<td>12000</td>
<td>6500</td>
<td>500</td>
</tr>
</tbody>
</table>

• \( T(\text{Employee}) = 25000 \), but now we know the distribution
Histograms

Ranks(rankName, salary)
- Estimate the size of Employee \( p_{\text{Salary}} \) Ranks

<table>
<thead>
<tr>
<th>Employee</th>
<th>0.25k</th>
<th>20k-40k</th>
<th>40k-60k</th>
<th>60k-80k</th>
<th>80k-100k</th>
<th>&gt; 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
<td>800</td>
<td>800</td>
<td>1200</td>
<td>1500</td>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ranks</th>
<th>0.25k</th>
<th>20k-40k</th>
<th>40k-60k</th>
<th>60k-80k</th>
<th>80k-100k</th>
<th>&gt; 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>20</td>
<td>40</td>
<td>80</td>
<td>100</td>
<td>2</td>
</tr>
</tbody>
</table>

Histograms

\( V(\text{Employee, Salary}) = 200 \)
\( V(\text{Ranks, Salary}) = 250 \)

\( p_{\text{Salary}} \) Ranks = Employee \( p_{\text{Salary}} \) Ranks \( \cup \ldots \cup \) Employee \( p_{\text{Salary}} \) Ranks

- A tuple \( t \) in Employee, joins with so many tuples in Ranks:
  - \( T(\text{Employee}) \times T(\text{Employee}) \times T(\text{Employee})/250 \)
- Then \( T(\text{Employee}) \times T(\text{Employee}) \times T(\text{Employee}) = \sum_{i=1}^{3} \frac{T_i}{250} \)
- \( = \frac{600x8 + 800x20 + 5600x40}{250} + \frac{12000x80 + 6500x100 + 500x2}{250} \)
- \( = \ldots \)

Recovery

<table>
<thead>
<tr>
<th>Type of Crash</th>
<th>Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong data entry</td>
<td>Constraints and Data cleaning</td>
</tr>
<tr>
<td>Disk crashes</td>
<td>Redundancy: e.g. RAID, archive</td>
</tr>
<tr>
<td>Fire, theft, bankruptcy…</td>
<td>Buy insurance, Change jobs…</td>
</tr>
<tr>
<td>System failures: e.g. power</td>
<td>DATABASE RECOVERY</td>
</tr>
</tbody>
</table>

Most frequent

System Failures

- Each transaction has _internal state_
- When system crashes, internal state is lost
  - Don’t know which parts executed and which didn’t
- Remedy: use a log
  - A file that records every single action of the transaction

Transactions

A *transaction* = piece of code that must be executed atomically
- In ad-hoc SQL
  - one command = one transaction
- In embedded SQL
  - Transaction starts = first SQL command issued
  - Transaction ends =
    - COMMIT
    - ROLLBACK (=abort)

Transactions

- Assumption: the database is composed of *elements*
  - Usually 1 element = 1 block
  - Can be smaller (=1 record) or larger (=1 relation)
- Assumption: each transaction reads/writes some elements
Primitive Operations of Transactions

- INPUT(X)
  - read element X to memory buffer
- READ(X,t)
  - copy element X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to element X
- OUTPUT(X)
  - write element X to disk

The Log

- An append-only file containing log records
- Note: multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Redo some transaction that didn’t commit
  - Undo other transactions that didn’t commit
- Three kinds of logs: undo, redo, undo/redo

Undo Logging

Log records
- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has committed
- <ABORT T>
  - T has aborted
- <T,X,v>
  - T has updated element X, and its old value was v

Undo-Logging Rules

U1: If T modifies X, then <T,X,v> must be written to disk before X is output to disk
U2: If T commits, then <COMMIT T> must be written to disk only after all changes by T are output to disk
- Hence: OUTPUTs are done early, before the transaction commits
Recovery with Undo Log

After system’s crash, run recovery manager

- Idea 1. Decide for each transaction T whether it is completed or not
  - <START T>....<COMMIT T>.... = yes
  - <START T>....<ABORT T>.... = no
  - <START T>........................ = no

- Idea 2. Undo all modifications by incomplete transactions

Recovery with Undo Log

Recovery manager:

- Read log from the end; cases:
  - <COMMIT T>: mark T as completed
  - <ABORT T>: mark T as completed
  - <T,X,v>: if T is not completed then write X=v to disk else ignore
  - <START T>: ignore

Recovery with Undo Log

... <T6,X6,v6> ...
... <START T6> ...
... <START T4> ...
... <T1,X1,v1> ...
... <T5,X5,v5> ...
... <T4,X4,v4> ...
... <COMMIT T5> ...
... <T3,X3,v3> ...
... <T2,X2,v2> ...

crash

Recovery with Undo Log

Question 1 in class:
Which updates are undone?

Question 2 in class:
How far back do we need to read in the log?

Recovery with Undo Log

- Note: all undo commands are idempotent
  - If we perform them a second time, no harm is done
  - E.g. if there is a system crash during recovery, simply restart recovery from scratch

Checkpointing

Checkpoint the database periodically

- Stop accepting new transactions
- Wait until all current transactions complete
- Flush log to disk
- Write a <CKPT> log record, flush
- Resume transactions

Recovery with Undo Log

When do we stop reading the log?

- We cannot stop until we reach the beginning of the log file
- This is impractical
- Better idea: use checkpointing
Undo Recovery with Checkpointing

During recovery, Can stop at first <CKPT>

\[ \ldots \]

\[ \ldots, <T9,X9,a1> \]

\[ \ldots \]

(all completed)

\[ <\text{CKPT}> \]

\[ \langle \text{START T2} \rangle \]

\[ \langle \text{START T3} \rangle \]

\[ \langle \text{START T5} \rangle \]

\[ \langle \text{START T4} \rangle \]

\[ <T1,X1,a1> \]

\[ <T5,X5,a5> \]

\[ <T4,X4,a4> \]

\[ <\text{COMMIT T5}> \]

\[ <T3,X3,a3> \]

\[ <T2,X2,a2> \]

other transactions

Transactions T2, T3, T4, T5

Nonquiescent Checkpointing

- Problem with checkpointing: database freezes during checkpoint
- Would like to checkpoint while database is operational
- Idea: nonquiescent checkpointing

Quiescent = being quiet, still, or at rest; inactive
Non-quiescent = allowing transactions to be active

Nonquiescent Checkpointing

- Write a <START CKPT(T1,\ldots,Tk)> where T1,\ldots,Tk are all active transactions
- Continue normal operation
- When all of T1,\ldots,Tk have completed, write <END CKPT>

Q: why do we need <END CKPT>?