Algorithms for Join Operations

• The join operation and its variants are the most time consuming operations in query processing.

• Most joins are either natural joins or equi-joins.

• Joins which involve two relations are called two-way joins while joins involving more than two relations are called multiway joins.

• While there are several different strategies that can be employed to process two-way joins, the number of potential strategies grows very rapidly for multiway joins.
Two-way Join Strategies

• We’ll assume that the relations to be joined are named R and S, where R contains an attribute named A and S contains an attribute named B which are join compatible.

• For the time-being, we’ll consider only natural or equijoin strategies involving these two attributes.

• Note that for a natural join to occur on attributes A and B, a renaming operation on one or both of the attributes must occur prior to the natural join operation.

  – Note too, that if attributes A and B are the only join compatible attributes in R and S, that the equi-join operation \( R \ast_{A=B} S \) has the same effect as a natural join operation.
Algorithms for Two-way Join Operations

- **(J1-nested loop):** A brute force technique where for each record \( t \in R \) (outer loop) retrieve every record \( s \in S \) (inner loop) and test if the two records satisfy the join condition, namely does \( t.A = s.B \)?

- **(J2-single loop w/access structure):** If an index or hash key exists for one of the two join attribute, for example, \( B \in S \), retrieve each record \( t \in R \) one at a time and then use the access structure to retrieve directly all matching records \( s \in S \) that satisfy \( t.A = s.B \).

- **(J3-sort-merge join):** If the records of both \( R \) and \( S \) are physically sorted (ordered) by the values of the join attributes \( A \) and \( B \), then the join can be processed using the most efficient strategy. Both relations are scanned in the order of the join attributes; matching the records that have the same \( A \) and \( B \) values. In this fashion, each relation is scanned only once.

- **(J4-hash-join):** In this technique, the records of both relations \( R \) and \( S \) are hashed using the same hashing function (on the join attributes) to the same hash file. A single pass through the smaller relation will hash its records to the hash file. A single pass through the other relation will hash its records to the same bucket as the first pass combining all similar records.
Pipelining Operations

- Query optimization can also be effected by reducing the number of intermediate relations that are produced as a result of executing a query stream.

- This reduction in the number of intermediate relations is accomplished by combining several relational operations into a single pipeline of operations. This method is also sometimes referred to as stream-based processing.

- While the combining of operations in a pipeline eliminates some of the cost of reading and writing intermediate relations, it does not eliminate all reading and writing costs associated with the operations nor does it eliminate any processing.

- As an example, consider the natural join of two relations R and S, followed by the projection of a set of attributes from the join result.
Pipelining Operations (cont.)

• In relational algebra this query looks like: $\pi_{(a, b, c)}(R \times S)$

• This set of two operations could be executed as:
  – construct the join of R and S, save as intermediate table T1. [T1 = R \times S]
  – project the desired set of attributes from table T1. [result = $\pi_{(a, b, c)}(T1)$]

• In the pipelined execution of this query, no intermediate relation T1 is produced. Instead, as soon as a tuple in the join of R and S is produced it is immediately passed to the projection operation to processing. The final result is created directly.

• In the pipelined version, results are being produced even before the entire join has been processed.
Pipelining Operations (cont.)

- There are two basic strategies that can be used to pipeline operations.
  - **Demand-driven pipelining**: In effect, data is “pulled-up” the query tree as operations request data to operate upon.
  - **Producer-driven pipelining**: In effect, data is “pushed-up” the query tree as lower level operations produce data which is set to operations higher in the query tree.
Demand-Driven Pipelining Example

**π_{s#}**  
Projection requests data from join operation

**π_{p#}**  
Join requests tuple from projection (below) and a tuple from SPJ

**σ_{color = red}**  
Projection requests tuple from selection

**P**  
Selection extracts tuple from P, if match tuple is set up the tree, if not, it is ignored
Producer-Driven Pipelining Example

As soon as first tuple arrives here from join a result is produced

As soon as first tuple is produced here is is sent to projection

As soon as first tuple is produced here is is sent to join

As soon as first tuple is produced here is is sent to projection
Using Heuristics in Query Optimization

• The parser of the high-level query language generates the internal representation of the query which is optimized according to heuristic rules.

• The access routines which execute groups of operations together are based upon the access paths available for the relations involved are chosen by the query optimizer.

• One of the main heuristic rules is to apply projections and selections as early as possible. This is useful because the size of the relations involved in subsequent join operations (or other binary operations) are as small as possible.

• Basically, the query optimizer generates several different query expressions and selects the best choice.
Using Heuristics in Query Optimization (cont.)

• When an equivalent query expression is generated, you must be
certain that it is in fact an equivalent expression.

• To this end, the query optimizer must follow certain transformation
rules that will ensure equivalency amongst the various query
expressions.

• The level of information available to the optimizer will affect the
effectiveness of the equivalence generation scheme.
  – At the lowest level – only relation names are known:
    • \( R \cap R \equiv R \)
    • \( \pi_X (R \cup S) \equiv \pi_X (R) \cup \pi_X (S) \)
    • \( \sigma_{A=B \text{ AND } B=C \text{ AND } A=C} (R) \equiv \sigma_{A=B \text{ AND } B=C} (R) \)
Using Heuristics in Query Optimization (cont.)

– If schema information is available:
  • Given $R(A, B)$, $S(B,C)$ with $r(R)$ and $s(S)$ then,
    • $\sigma_{A=a}(R \ast S) \equiv \sigma_{A=a}(R) \ast S$

– If constraint information is known, they provide even more information and modification possibilities:
  • If you know that $R(A,B,C,D)$ with $r(R)$ and you also know that $r$ satisfies $B \rightarrow C$, then $\pi_{A,B}(r) \ast \pi_{B,C}(r) \equiv \pi_{A,B,C}(r)$
  • In general, there are many different equivalences that will hold and the optimizer can utilize as many as possible.
    – For example: $r \cup r \equiv r$, $r \cap r \equiv r$, $r - r \equiv \emptyset$
Using Heuristics in Query Optimization (cont.)

• Commutivity rules can also be applied to optimize query execution.

• For example what is the difference between R * S and S * R?
  
  – Suppose that R contains 3 tuples and S contains 5 tuples. Further suppose that each tuple in R is 10 bytes long and each tuple in S is 100 bytes long.
  
  – R * S: 1 pass through R generates $3 \times 10$ bytes = 30 bytes. Three passes through S (one for each tuple generated from R) generates $15$ tuples $\times$ 100 bytes = 1500 bytes. Total = 1530 bytes.
  
  – S * R: 1 pass through S generates $5 \times 100$ bytes = 500 bytes. Five passes through R (one for each tuple generated from S) generates $15$ tuples $\times$ 10 bytes = 150 bytes. Total = 650 bytes.
  
  – Clearly, S*R is a better strategy than is R*S.
Cost estimation is typically only used for “canned” query execution code, i.e., compiled queries that will be executed repeatedly.

The time and effort required for this type of analysis is not justified for simple one-time query execution.

The cost estimation technique considers the cost of executing a query from four different perspectives:

1. Access costs to secondary storage: this involves all the costs of searching, reading, and writing secondary storage.
2. Storage costs: this involves the cost of storing the intermediate files generated by the chosen execution strategy.
3. Computation costs: Sorting, merging, computation in attributes (selection and join conditions).
4. Communication costs: In a distributed environment, this includes the cost of shipping the query and/or its results to the originating site.
Semantic Query Optimization

- This technique uses the semantics of the database and the various constraints that apply to semantically modify queries into queries which are more efficient.

- For example, suppose a user issues the following query:
  \[ \pi_{s\#} (\sigma_{qty > 100} (SPJ)) \]  
  \{list supplier numbers for suppliers who ship at least one part in a quantity greater than 100.\}

- If a constraint exists that states: all quantities \( \leq 75 \), then the optimizer could inform the system that the query did not need to be executed at all and the result is simply the empty set.
Introduction to Transaction Processing

• The execution of any “program” that either accesses (queries) or changes the database contents is called a transaction.

• Serial transactions – two or more transactions are processed in serial fashion with one transaction starting and completing before the next transaction begins execution. At no time, is more than one transaction processing or making progress.

• Interleaved transactions – two or more transactions are processed concurrently with only one transaction at a time actually making progress. This most often occurs on a single multi-programmed CPU.

• Simultaneous transactions – two or more transactions are processed concurrently with any number progressing at one time. This is a multiple CPU situation.
Serial transactions (unknown number of CPUs)
Introduction to Transaction Processing (cont.)

Interleaved transactions (single CPU)
Introduction to Transaction Processing (cont.)

Simultaneous transactions (3 CPUs shown)
Introduction to Transaction Processing (cont.)

• When viewed at the transaction level, any transaction has the potential to access the database in two ways:
  – read(item): reads the value of some database item.
  – write(item): write the value of an item into the database.

• These are not atomic operations.

• To read an item the following must occur:
  – find the address of the disk block that contains the item.
  – copy the disk block into buffer (if not already present).
  – copy the item from the buffer into the “program”.
• To write an item the following must occur:
  – find the address of the disk block that contains the item.
  – copy the disk block into buffer (if not already present).
  – copy the item from the buffer into the “program”.
  – store the updated block from the buffer back onto the disk (at some point in time, usually not immediately).

• When to write back is typically up to the recovery system of the database and may involve OS control.

• Too early of a write back may cause unnecessary data transfers.

• Too late of a write back may cause unnecessary blocking.
Concurrency Control

- Given a consistent (correct?) state of the database as input an individually correct transaction will produce a correct state of the database as output, if that transaction is executed in isolation.

- The goal of concurrency control is to allow multiple transactions to be processing simultaneously within a certain time period with all of the concurrent transactions producing a correct state of the database at then end of their concurrent execution.
**Concurrency Control – Why It's Needed**

- There are many different types of conflicts that can occur between concurrently executing processes if concurrency control is not enforced.

**Lost Update Problem**

- Suppose two distinct transactions T1 and T2 are processing in the concurrent order shown below accessing a common value *n*.

<table>
<thead>
<tr>
<th>time</th>
<th>action</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>T1 performs read(n)</td>
<td>suppose T1 reads value of <em>n</em> = 5</td>
</tr>
<tr>
<td>t1</td>
<td>T2 performs read(n)</td>
<td>T2 will read a value of <em>n</em> = 5</td>
</tr>
<tr>
<td>t2</td>
<td>T1 performs write(n-1)</td>
<td>T1 will write value of <em>n</em> = 4</td>
</tr>
<tr>
<td>t3</td>
<td>T2 performs write(n-1)</td>
<td>T2 will also write value of <em>n</em> = 4!</td>
</tr>
</tbody>
</table>

- Problem: The update performed by T1 at time t2 is “lost” since the update written by T2 at time t3 overwrites the previous value.
Handling the Lost Update Problem

• There are several different ways in which the lost update problem can be handled.

  1. Prevent T2 from reading the value of n at time t1 on the grounds that T1 has already read the value of n and may therefore update the value.

  2. Prevent T1 from writing the value of n-1 at time t2 on the grounds that T2 has also read the same value of n and would therefore be executing on an obsolete value of n, since T2 cannot re-read n.

  3. Prevent T2 from writing the value of n-1 at time t3 on the grounds that T1 has already updated the value of n and since T1 preceded T2, then T2 is using an obsolete value of n.

• The first two of these techniques can be implemented using locking protocols, while the third technique can be implemented with time-stamping. We’ll see both of these techniques later.
The Dirty Read Problem

Dirty Read Problem

- Suppose two distinct transactions T1 and T2 are processing in the concurrent order shown below accessing a common value $n$.

<table>
<thead>
<tr>
<th>time</th>
<th>action</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>T1 performs read($n$)</td>
<td>suppose T1 reads value of $n = 5$</td>
</tr>
<tr>
<td>t1</td>
<td>T1 performs write($n-1$)</td>
<td>T1 writes a value of $n = 4$</td>
</tr>
<tr>
<td>t2</td>
<td>T2 performs read($n$)</td>
<td>T2 will read value of $n = 4$</td>
</tr>
<tr>
<td>t3</td>
<td>T1 aborts</td>
<td>T2 is executing with a “bad” value of $n$</td>
</tr>
</tbody>
</table>

- Problem: T2 is operating with a value that was written by a transaction that aborted prior to the completion of T2. When T1 aborts all of its updates must be undone, which means that T2 is executing with a bad value of $n$ and therefore cannot leave the database in a consistent state.

Solution: T2 must also be aborted.
The Unrepeatable Read Problem

Unrepeatable Read Problem

- Suppose two distinct transactions T1 and T2 are processing in the concurrent order shown below accessing a common value \( n \).

<table>
<thead>
<tr>
<th>time</th>
<th>action</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>T1 performs read(n)</td>
<td>suppose T1 reads value of (n = 5)</td>
</tr>
<tr>
<td>t1</td>
<td>T1 performs read(n)</td>
<td>T1 reads a value of (n = 5)</td>
</tr>
<tr>
<td>t2</td>
<td>T2 performs write(n-1)</td>
<td>T2 will write value of (n = 4)</td>
</tr>
<tr>
<td>t3</td>
<td>T1 performs read(n)</td>
<td>T1 reads a different value of (n) this time</td>
</tr>
</tbody>
</table>

- Problem: When T1 performs its second read of \(n\), the value is not the same as its first read of \(n\). T1 cannot repeat its read.

Solution: This problem is typically handled with locking which is rather inflexible, but can also be solved with time-stamping.
The Transaction Recovery System

- Whenever a transaction is submitted to the DBMS for execution, the DBMS is responsible for making sure that either:
  1. All operations of the transaction are completed successfully and their effect is permanently recorded in the database, or
  2. The transaction has no effect whatsoever on the database or any other transaction.

- If a transaction fails after executing some of its operations, problems will occur with consistency in the database. Therefore, if a transaction fails after its is initiated but prior to its commitment, all of the effects of that transaction must be undone from the database.
The Transaction Recovery System (cont.)

- Types of failures for a transaction:
  - System crash – some type of hardware or system failure occurs.
  - Transaction error – integer overflow, division by zero, operator intervention.
  - Local errors or exception conditions – required data is not available.
  - Concurrency control enforcement – serializability is violated, deadlock detection victim selection, etc.
  - Disk errors – error correction/detection.
  - Physical problems – fire, power failure, operator error, etc.
The States of a Transaction

• A transaction can be in one of several different states:

  – **begin_transaction**: marks the beginning of the transaction.
  
  – **read/write**: specifies the various db operations performed by the transaction.
  
  – **end_transaction**: specifies that all read/write operations have ended and the transaction is ready to terminate. Note: this does not actually end the transactions time in the system – now it heads to the concurrency control system for verification.
  
  – **commit**: marks the successful end of the transaction – its effects are now permanent (committed) in the database and cannot be undone.
  
  – **abort (rollback)**: marks the unsuccessful end of the transaction. All changes and effects in the database must be undone and/or other transactions must be aborted. No changes are committed for the transaction.
The States of a Transaction (cont.)

- **active**
  - begin_transaction
  - read/write
- **partially committed**
  - end_transaction
  - abort
- **failed**
  - abort
- **committed**
  - commit
- **terminated**
  -