CT3532 Database Systems Lecture 1

Dr. Colm O’Riordan

School of Computer Science

Contact Details

Colm O’ Riordan

Email: colm.oriordan@universityofgalway.ie

Details

Lecture slides to be made available weekly on canvas.

Discussion boards for any queries people may have.

Short regular exercise sheets (not graded) to help/guide study.

Assignments during the semester

Recommended Texts

Fundamentals of Database Systems by Elmasri and Navathe 005.74 ELM

Database system concepts by Silberschatz, A. 005.74 SIL

Plenty of other good database books in library – (e.g. Date and by Ullman)

Course Grading

- Assignments account for 30%

- Plagiarism of assignment work not permitted. Strictly enforced.

- Remainder awarded in examination

DESIGN

Design by synthesis

Functional dependencies

Armstrong’s axioms

Closure and cover set

Algorithm for generating design

Normal forms

Non-additive join property

DESIGN

Physical Design

Indexing strategies

Denormalisation

Choice of keys

QUERY PROCESSING

Indexing

Indexing fundamentals

B-trees; B+trees

Extendible Hashing

QUERY PROCESSING

Query execution and optimisation

Relational algebra and SQL

Heuristic optimisation

Algorithms for SQL operators

QUERY PROCESSING

Algorithms for SQL operators

Algorithms for select, project, set operators, join

Indexed based approaches

Hashing approaches

Sorting methods

Parallel approaches

TRANSACTIONS

Transactions

Properties

States

Schedules

Properties of schedules

Serializability

TRANSACTIONS

Concurrency Control

locking techniques

2 phase locking

time-stamping

multi-version timestamp control

TRANSACTIONS

Recovery

Immediate update and deferred update protocol

Algorithms for recovery under these protocols

Models

- Object Relational
- Object Oriented Databases
- Comparison of Models

Models

- NOSQL database
- Types
- Languages and models
- Advantages/Limitations

Models

- Distributed Databases
- query processing
- concurrency control (locking and time-stamping)
- recovery

Models

- Parallel Databases
- approaches
- parallel algorithms for relational operators

Models

- Graph Databases
- structure
- languages

CT3532 Database Systems

Colm O’Riordan

Recap - ER modelling

- ER modelling concepts
- Guidelines to map to a relational model

Recap - Normalisation

Normalisation can be used to:

- develop a normalised relational schema given the universal relation
- verify correctness of relational schema developed from conceptual design.

Decompose relation such that it satisfies successively restrictive normal forms

Desirable properties of a relational schema

- Clear semantics of a relation
- Reducing the number of redundant values in tuples
- Reducing the number of null values in tuples
- Disallowing the possibility of generating spurious tuples.

Clear semantics

The semantics of a relation refers to how the attributes grouped together in a relation are to be interpreted.

If ER modelling is done carefully and the mapping undertaken correctly, it is likely the semantics of the resulting relation will be clear.

One should try to design a relation so that it is easy to explain its meaning.

Reducing redundancy

The presence of redundancy leads to:

1. waste of storage space
2. potential for anomalies (deletion, update, insertion). One try to design relations so that no anomalies may occur. If an anomaly can occur, it should be noted.
3. Normalisation will remove many of the potential anomalies.

Reducing the number of null values

Having nulls is often necessary; however having nulls can create problems.

- waste of space
- different interpretations:
  - attribute does not apply to this tuple
  - attribute value is unknown
  - attribute value is known but absent.
- difficulty with aggregate functions
- different meanings with respect to different join operations.

Disallowing generation of spurious tuples

If a relation R is decomposed into R_1 and R_2 and connected via a primary key - foreign key pair, then performing an equi-join between R_1 and R_2 on the involved keys should not produce tuples that were not in the original relation R.

Desirable properties - more formally

Typically we have a relation, R, and a set of functional dependencies, F, defined over R.

We wish to create a decomposition:

$$D = \{R_1, R_2, \ldots, R_n\}$$

We wish to guarantee certain properties of this decomposition.

Desirable properties - more formally

We require that all attributes in the original R be maintained in the decomposition. i.e.,

$$R = R_1 \cup R_2 \cup \ldots \cup R_n$$

Normalisation - Recap

- A relation R is said to be in first normal form if there are no repeating fields.
- A relation R is said to be in 2NF if it is in 1NF and if every non-prime attribute is fully functionally dependent on the key.
- A relation is said to be in third normal form (3NF) if it is in 2NF and if no non-prime attribute is transitively dependent on the key.

Boyce-Codd Normal Form

A relation is said to be in Boyce-Codd Normal form (BCNF) if the relation is in *3NF* and *if every determinant is a candidate key*.

Sample data

StudentNo	Major	Advisor
123	I.T.	Smith
123	Econ	Murphy
444	Biol.	O’ Reilly
617	I.T.	Jones
829	I.T.	Smith

Constraints:

- A student may have more than one major
- For each major a student has only one advisor
- Each major can have several advisors
- Each advisor advises one major
- Each advisor can advise several students

Functional Dependencies

- {StudentNo, Major} → {Advisor}
- {Advisor} → Major

Update anomaly may exist

If student 444 changes major, we lose information that O' Reilly supervises Biology

Decompose tables so as to satisfy BCNF

TAKES: StudentNo, Advisor
ADVISES: Advisor, Major

General Rule

Consider relation R with functional dependencies F.
If $X \rightarrow Y$ violates BCNF, decompose R into

- $\{R - Y\}$
- $\{XY\}$.

Let $R = \{A, B, C, D, E, F, G, H\}$

The functional dependencies defined over R are:

$$
\begin{aligned}
&A \rightarrow D\\
&B \rightarrow E\\
&E \rightarrow F\\
&F \rightarrow G\\
&F \rightarrow H\\
&\{A, B\} \rightarrow C\\
&C \rightarrow A
\end{aligned}
$$

Decompose R such that BCNF is satisfied.

Design by synthesis

Colm O'Riordan

School of Computer Science

Design by Synthesis - Background

Typically, we have the relation R and a set of functional dependencies F.

We wish to create a decomposition $D = R_1, R_2, ...R_m$.

Clearly, all attributes of R must occur in a least one schema R_i, i.e.,

$$U_{i=1}^{m} R_i = R$$

This is known as the **attribute preservation** constraint.

Functional dependencies

A functional dependency is a constraint between two sets of attributes. A functional dependency $X \rightarrow Y$ exists if for all tuples t_1 and t_2, if $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$.

Usually only specify the obvious functional dependencies. There may exist many more.

Given a set of functional dependencies F, the closure of F (denoted F^+) refers to all dependencies that can be derived from F.

A set of inference rules exist, that allow us to deduce or infer all functional dependencies from a given initial set.

Known as Armstrong’s Axioms

Armstrong's Axioms

- Reflexivity: if $X \supseteq Y$, then $X \rightarrow Y$
- Augmentation: if $X \rightarrow Y$, then $XZ \rightarrow YZ$
- Transitivity: if $X \rightarrow Y$, $Y \rightarrow Z$, then $X \rightarrow Z$
- Projectivity: if $X \rightarrow YZ$, then $X \rightarrow Z$
- Additivity: if $X \rightarrow Y, X \rightarrow Z$, then $X \rightarrow YZ$
- Pseudo-transitivity: if $X \rightarrow Y$, $WY \rightarrow Z$, then $WX \rightarrow Z$

The first three rules have be shown to be *sound* and *complete*.

Sound

Given a set F specified on a relation R, any dependency we can infer from F using the first three rules, holds for every state r of R that satisfies the dependencies in F.

Complete

We can use the first three rules repeatedly to infer all possible dependencies that be can be inferred from F.

For any set of attributes A, we can infer A^+, the set of attributes that are functionally determined by A given a set of functional dependencies.

Algorithm to determine the closure of *A* under *F*

```
$A^+$ := $A$;
repeat
$oldA+$ := $A^+$
for each functional dependency $Y \rightarrow Z \in F$ do
        if $A^+ \supseteq Y$, then
                $A^+$ := $A^+ \cup Z$
until ($A^+$ == $oldA^+$)
```

Cover Sets

A set of functional dependencies, F, *covers* a set of functional dependencies E, if every functional dependency in E is in F^+

Equivalence

Two set of functional dependencies, E and F are equivalent is $E^+ = F^+$

We can check if F covers E by calculating A^+ with respect to F for each functional dependency $A \rightarrow B$ and then checking that A^+ includes the attributes of B

Minimal Cover Sets

A set of functional dependencies, F, is minimal if:

- Every functional dependency in F has a single attribute for its right hand side.
- We cannot remove any dependency from F and maintain a set of dependencies equivalent to F.
- We cannot replace any dependency $X \rightarrow A$ with a dependency $Y \rightarrow A$ where $Y \subset X$, and still maintain a set of dependencies equivalent to F.

All functional dependencies $X \rightarrow Y$, specified in F, should exist in one of the schema R_i, or should be inferrable from the dependencies in R_i.

This is known as the **dependency preservation** constraint.

Each functional dependency specifies some constraint; if the dependency is absent then some desired constraint is also absent.

If a functional dependency is absent then we must enforce the constraint in some other manner. This can be inefficient.

Given F and R, the *projection* of F on R_i, denoted $\pi_{R_i}(F)$ where R_i is a subset of R, is the set $X \rightarrow Y$ in F^+ such that attributes $X \cup Y \in R_i$.

A decomposition of R is dependency-preserving if $((\pi_{R_1}(F)) \cup \ldots \cup (\pi_{R_m}(F)))^+ = F^+$.

Theorem:

It is always possible to find a decomposition D with respect to F such that:

1. the decomposition is dependency-preserving
2. all R_i in D are in 3NF

We can always guarantee a dependency-preserving decomposition to 3NF.
Algorithm:

1. Find a minimal cover set G for F.
2. for each left hand side X of a functional dependency in G, create a relation $X \cup A_1 \cup A_2 \ldots A_m$ in D, where $X \rightarrow A_1 X \rightarrow A_2 \ldots$ are the only dependencies in G with X as a left hand side.
3. Group any remaining attributes into a single relation.

Lossless joins

Consider the following relation:
EMPPROJ: ssn, pnumber, hours, ename, pname, plocation

and its decomposition to:

EMPPROJ1: ename, plocation
EMPLOCAN: ssn, pno, hrs, pname, plocation

If we perform a natural join on these relations, we may generate spurious tuples.

Lossless Joins

Also known as *non-additive joins*.

When a natural join is issued against relations, no spurious tuples should be generated.

A decomposition $D = \{R_1, R_2, \ldots R_n\}$ of R has the lossless join property wrt to F on R if for every instance r the following holds:

$$\bowtie (\pi_{R_1}(r), \ldots \pi_{R_m}(r)) = r)$$

We can automate procedure for testing for lossless property.

Can also automate the decomposition of R into $R_1, \ldots R_m$ such that it possesses the lossless join property.

A decomposition $D = \{R_1, R_2\}$ has the lossless property iff:

- functional dependency $(R1 \cap R_2) \rightarrow \{R_1 - R_2\}$ is in F^+
- or functional dependency $(R1 \cap R_2) \rightarrow \{R_2 - R_1\}$ is in F^+

Furthermore, if a decomposition has the lossless property, and we decompose one of R_i such that this also is a lossless decomposition, then replacing that decomposition of R_i in the original decomposition will result in a lossless decomposition.

Algorithm to decompose to BCNF

```
Let $D = R$
while there is a schema B in D that violates BCNF do
  choose B
  find functional dependency $(X \to Y)$ that violates BCNF
  replace B with
   $(B - Y)$ and $(X \cup Y)$
```

So, we guarantee a decomposition such that:

- all attributes are preserved
- lossless join property is enforced
- all R_i are in BCNF

It is not always possible to decompose R into a set of R_i such that all R_i satisfy BCNF and properties of lossless joins and dependency preservation are maintained.

We can guarantee a decomposition such that:

- all attributes are preserved
- all relations are in 3NF
- all functional dependencies are maintained
- the lossless join property is maintained

Algorithm: Finding a key for relation schema R

```
set K := R.
For each attribute A $\in$ K.
    compute $(K-A)^+$ wrt to set of functional dependencies.
    if e $(K-A)^+$ contains all the attributes in R, the set K := K - {A}.
```

Summary

Given a set of functional dependencies F, we can develop a minimal cover set.
Using this we can decompose R into a set of relations such that all attributes are preserved, all functional dependencies are preserved, the decomposition has the lossless join property and all relations are in 3NF.

Advantages

- Provides a good database design.
- Can be automated.

Disadvantages

- Oftentimes, numerous good designs are possible.

CT3532 Database Systems - Further Design

Colm O'Riordan

We have seen that through design by synthesis we can obtain a *good* design.

This guarantees that the final design schema exhibits certain desirable features

However, occasionally we wish to violate the above guidelines to improve performance.

We must pay attention to transaction requirements.

Try to:

- Identify future required transactions
- their relative frequency
- the required response time

Indexing strategies

The information gleaned from the above analysis can inform the design of the indexing strategy.

Usually place index on fields that are to be frequently queried upon.

Choice of index (primary, secondary, B-tree, B+-tree, clustering) will depend on:

- type of query expected
- types permitted by DBMS
- type of field involved

Also try to identify which tables will be joined frequently and on which attributes. Common to build indexes on these attributes.

Key choice

Performance requirements can lead to a change in the logical design.
Consider a table containing 1000 employees. The SSN number was chosen in the conceptual design as a choice of key.

Now consider the performance requirements involve joining this table to another table which has SSN as a foreign key.

This query is expected to be very frequent with a short short response time required.

SSN numbers are 8 characters long. Any index built on this will contain index values of 8 characters wide.

We only have 1000 employees - could introduce surrogate key (emp id) and use this for indexes to handle joins.

Denormalisation

Denormalisation is the process of making compromises to the normalised tables by introducing intentional redundancy for performance reasons.

Decisions regarding denormalisation are made during the transaction requirements analysis.

Denormalisation involves a tradeoff between:

- introducing redundancy and potential for anomalies
- increased efficiency of certain transactions

Downward Denormalisation

Consider the following relations:

CUSTOMER: ID, address, name, telephone
ORDER: orderno, date, date invoice, cus ID

Assume that the queries of the following type are extremely frequent and require a fast response time:

```
SELECT ID, name
FROM CUSTOMER, ORDER
WHERE CUSTOMER.ID = ORDER.cus_ID
AND orderno = 46;
```

Main cost in evaluating this query is the join operation. We can avoid this costly join be adding the `name` field to **ORDER** table.

This gives us:

ORDER: <u>orderno</u>, date, date invoice, cus ID, name

We have now introduced redundancy (violates 3NF), which leads to potential update, delete and insert anomalies. However, queries of the type above can be dealt with more efficiently.

Other types of denormalisation exist:

- Upward denormalisation: store aggregate of values from one table in another
- Intra-table denormalisation: explicitly store information in a table that can be derived from other attributes

BTrees and B+Trees

September 26, 2023

Generalised Search Tree

Each node has format:

$$P_1, K_1, P_2, K_2, ...P_{n1}, K_{n1}, P_n$$

where P_i are tree values and K_i are search values.
In a database context, a node corresponds to a disk block.
Hence, the number of values per node depends on the size of the key field, block size and block pointer size.

Constraints

The following constraints hold:

- $K_1 < K_2 < ... < K_{n1} < K_n$
- For all values x in a subtree pointed to P_i, $K_{i1} < x < K_i$
- The number of tree pointers per node is known as the order $/rho$ of the tree

Efficiency

- For a generalised search tree:

$$T(N) = 1 + T(N/) = ... = O(log(N))$$

- This assumes a balanced tree
- In order to guarantee this efficiency in searching and in other operations, we need techniques to ensure a balanced tree

B Trees

- A B Tree is a balanced generalised search tree
- Can be viewed as a dynamic multi-level index
- The properties of a search tree still hold.
- The algorithms for insertion and deletion of values are modified in order

B Trees: Node structure

- The node structure contains a record pointer for each key value.
- Node structure is as follows:

$$P_1, < K_1, Pr_1 > P_2, < K_2, Pr_2 >, ...P_{n1}, < K_{n1}, Pr_{n1} >, P_n$$

Example

- Consider a B tree of order 3 (two values and 3 tree pointers per node/block).
- Insert records with key values: 10, 6, 8, 14, 4, 16, 19, 11, 21

Algorithm to insert value into B Tree

1. Find appropriate leaf level node to insert value
2. If space remains in leaf-level node, then insert the new value in correct location.
3. If no space remains, we need to deal with collision.

Dealing with collisions

1. split node into left and right nodes
2. propagate middle value up a level and place value in node there (*)
3. place values less than middle value in the left node
4. place values greater than the middle value in the right node

*

Note: this propagation may cause further propagations and even the creation of a new root node

- This maintains the balanced nature of the tree.
- $RightarrowO(log_{\rho}(N))$ for search, insertion and deletion
- However, there is always potential for unused space in the tree.
- Note: Empirical analysis has shown that B-trees remain 69

Exercises

- Can you define an algorithm for deletion (at a high level)?
- How much work is needed in the various cases (best, average, worst)?

B+ tree

- The most commonly used index type is the B+-tree - a dynamic, multi-level index.
- Differs from a B Tree in terms of structure.
- Insertion and deletion algorithms slightly more complicated.
- Offers increased efficiency over B Tree. Ensures a higher order ρ.
- Two different node structures in B+ Trees:
  - internal nodes
  - leaf-level nodes

Node structure

- All record pointers are maintained at the leaf level in a B+tree.
- internal node structure
- $$P_1, K_1, P_2, K_2, \ldots P_{n1}, K_{n1}, P_n$$
- No record pointers.
- Less information per record; hence more search values per node

Node structure

- leaf level node structure
- One tree pointer is maintained at each leaf-level node. This points to the next leaf-level node.
- Each node's structure $K_1, Pr_1, K_2, Pr_2, ...K_m, Pr_m, P_{next}$
- The P_{next} pointer facilitates range queries.
- Note only one tree pointer per node at the leaf-level.

Example

- Let B = 512,P = 6, Pr = 7,K = 10
- Assume 30000 records as before.
- Assume tree is 69% full.
- How many blocks will the tree require? How many block accesses will a search require?

Example

- A tree of order ρ has at most $\rho - 1$ search values per node
- For a B+ Tree, there are two types of tree nodes; hence there are 2 different orders ρ and ρ_{leaf}
- To calculate ρ:
- $$\rho(|P|) + (\rho - 1)(|K|) \leq B$$
- $$\Rightarrow 16\rho < 522$$
- $$\Rightarrow \rho = 32$$

- To calculate ρ_{leaf}
- $$|P| + (\rho_{leaf})(|K| + |Pr|) \leq B$$
- $$\Rightarrow 17(\rho_{leaf}) \leq 506$$
- $$\rho_{leaf} = 29$$

Given fill factor = 69%:

- Each internal node will have, on average, 22 pointers
- Each leaf level node will have, on average, 20 pointers

- Root: 1 node 21 entries 22 pointers
- level1: 22 nodes 462 entries 484 pointers
- level2: 484 nodes .. etc.
- leaf level: ....

- Hence, 4 levels are sufficient
- Number of block accesses = 4 + 1
- Number of block 1 + 22 + 484 + ..

Recap

- Looked at structure of a BTree;
- Looked at insertion algorithm for BTrees.
- Introduced a B+Tree and looked at some calculations in order to illustrate how to work out the required size and number of accesses.

Dynamic Hashing

Introduction

- Can we improve upon logarithmic searching?
- Hashing is a technique that attempts to provide constant time for searching and insertion, i.e O(K)
- The basic idea for both searching and insertion is to apply a hash function to the search field of the record.
- The return value of the hash function is used to reference a location in the hash table

‘

Different approaches

- Create a hash table containing N addressable ‘slots’
- Each “slot’ may contain one record
- Create a hash function that returns a value to be used in insertion and searching
- The value returned by the hash function must be in the correct range, i.e, the address space of the hash table
- If the range of the keys is that of the address space of the table, we can guarantee constant time lookup
- Usually, this is not the case as the address space of the table is much smaller than that of the search field

- With numeric keys can use modulo-division or truncation
- With character keys must first convert to integer value. Can achieve this by multiplying ASCII code of characters together and then applying modulo-division
- Cannot guarantee constant time performance as collisions will occur i.e., two records with different search values being hashed to the same location in the table
- we require a collision resolution policy

Efficiency then depends on the number of collisions. Number of collisions depends mainly on the load factor, λ, of the file:

$$\lambda = \frac{\text{no of records}}{\text{no of slots}}$$

Collision Resolution Policy

- **Chaining:** if location is full, add item to a linked list
- performance degrades if load factor is high.
- lookup time is, on average, 1 + λ (average case)
- **Linear Probing:** if location is full, check in a linear manner for next free space.
- This can degrade to a linear scan: performance:
- if successful: $0.5(1 + \frac{1}{1-\lambda})$
- if unsuccessful: $0.5(1 + \frac{1}{(1-\lambda)^2})$
- one big disadvantage is that this leads to the formation of clusters
- **Quadratic probing:** if location is full, check location $x + 1$, location $x + 4$, ... $(x + n)^2$
- less clustering
- **Double hashing:** if location x occupied, then apply second hash function can help guarantee even distribution (a fairer hash function)

Dynamic Hashing

- Care should be taken in designing hash function. Usually require fair hash function.
- Difficult to guarantee if no/limited information available about the type of data to be stored.
- Often heuristics can be used if domain knowledge available
- Can have both internal (some data structure in memory) or external hashing (to file locations)
- Size of original table or file?

- We considered hashing to an array (in memory).
- In reality, in database systems, we are typically hashing to a disk block (bucket) each of which can contain a fixed number of records.
- If a block is full, then we have a collision.
- Typically dealt with using overflow buckets (chaining).

- The cases we've considered thus far deal with the idea of a fixed hash table; this is referred to as static hashing.
- Problems arise if the database grows larger than planned; too many overflow buckets and performance degrades.
- A more suitable approach is dynamic hashing, where the table/file can be resized as needed.

General Approach

- use a family of hash functions h_0, h_1, h_2, etc.
- h_{i+1} is a refinement of h_i
- For example, $Kmod2^i$
- Develop a base hash function that maps key to a positive integer
- Then use, $h_0(x) = x\text{mod}2^b$ for a chosen b. There will be 2^b buckets initially.
- Can effectively double the size of the table by incrementing b

- Common dynamic hashing approaches: extendible hashing and linear hashing.
- Conceptually double the number of buckets when re-organising. From an implementation perspective, we do not actually double size as it may not be needed.
- Extendible hashing - reorganise buckets when and where needed
- Linear hashing - reorganise buckets when but not where needed.

Extendible Hashing

- When a bucket overflows, split that bucket in two.
- Conceptually, split all the buckets in two A directory (a form of index) is use to achieve this conceptual doubling.

Extendible Hashing

- If a collision or overflow occurs, we don't re-organise the file by doubling the number of buckets; too expensive.
- Instead we maintain a directory of pointers to buckets, we can effectively double the number of buckets by doubling the directory, splitting just the bucket that overflowed.
- As the directory is much smaller than file, so doubling it is much cheaper.

- On overflow, we split the bucket (allocate new bucket and re-distribute contents).
- We double the directory size if necessary.
- For each bucket, we maintain a local depth (effectively the number of bits needed to hash an item here).
- Also maintain a global depth for the directory; the number of bits used in indexing items.
- These values can be used to determine when to split the directory.

- If overflow in bucket with local depth = global depth, then split bucket, re-distribute contents, double the directory.
- If overflow into bucket with local depth < global depth, then split bucket, re-distribute contents. Increase local depth.
- If directory can fit in memory, then retrieval for point queries can be achieved with one disk read.

Linear Hashing

- Another approach to indexing to a dynamic file. Similar idea in that a family of hash functions are used ($h = K \bmod 2^i$), but differs in that no index is needed.
- Initially, create a file of M buckets. $K \bmod M^1$ is a suitable hash function.
- We will use a family of such functions $K \bmod (2^i \times M), i = 0$ initially.
- Can view the hashing as comprising a sequence of phases.
- For phase j, the hash functions $K \bmod 2^j \times M$ and $K \bmod 2^{j+1} \times M$ are used.

- Splitting a bucket means to redistribute the records into two buckets: the original one and a new one.
- In phase *j*, to determine which ones go into the original while the others go into the new one, we use $h_{j+1}(K) = Kmod2^{j+1} \times M$ to calculate their address.
- Irrespective of the bucket which causes the overflow, we always split the next bucket in a linear order.
- We begin with bucket 0, and keep track of which bucket to split next, *p*.
- At the end of a phase when *p* is equal to the number of buckets present at the start of the phase, we reset *p* and a new phase begins (*j* incremented).

Joins and Sorts

Join

Quite a few approaches/algorithms can be used

Nested Loop Join

To perform the join r $\bowtie$ s:

```
for each tuple t_r in r do
   for each tuple t_s in s do
      if t_r and t_s satisfy join condition
         add (t_r,t_s) to result
   end
end
```

Performance ..

- expensive approach
- every pair of tuples is checked to see if they satisfy join condition
- If one of the relations fits in memory, it is beneficial to use this in the inner loop. (known as the inner relation).

Block Nested Loop Join

Variation on the nested loop-join. Increases efficiency by reducing number of block accesses.

```
for each block B_r in r do
   for each block B_s in s do
      for each tuple t_r in B_r do
           for each tuple t_s in B_s do
              if t_r and t_s satisfy join condition
                  add (t_r,t_s) to result
           end
       end
   end
end
```

Indexed Nested Loop Join

If in a nested loop join, there is an index available for the inner table, replace file scans with index accesses

Merge Join

- If both relations are sorted on the joining attribute, then merge relations.
- Technique is identical to merging two sorted lists (like the merge step in a merge-sort algorithm)
- Much more efficient that a nested join
- Can also be computed for relations not ordered on a joining attribute, but have indexes on joining attribute
- Efficiency?

Hash Join

- Create a hashing function which maps the join attribute(s) to partitions in a range $1 \dots N$
- For all tuples in r, hash the tuples to H_{ri}
- For all tuples in s, hash the tuples to H_{si}
- For $i = 1$ to N, join partitions $H_{ri} = H_{si}$

Sorting

Sorting

Important operation because:

- if a query specifies ORDER BY
- used prior to relational operators (e.g. Join) to allow more efficient processing of operation

Can sort a relation:

- physically: - actual order of tuples re-arranged on disk
- logically: - build an index and sort index entries

Two main cases:

- where relation to be sorted fits in memory- can then use standard sorting techniques (e.g. quicksort)
- where relation doesn't fit in memory. The most common approach is to use external sort-merge

External Sort Merge - Step 1

```
i := 0;
repeat
   read M blocks of the relation
   sort M blocks in memory
   write sorted data to file Ri
until end of relation

M = number of page frames in main memory buffer
```

External Sort Merge - Step 2

Wish to merge the files from each run in step 1

```
read first block of each Ri into memory

repeat
choose first (in sort order) from pages
write tuple to output
remove tuple from buffer
if any buffer Ri if empty and not eof(Ri)
    read next block from Ri into memory
until all pages empty
```

effectively a N-way merge (extension of idea in the merge step of the merge sort algorithm)

Parallel Databases

- increased transaction requirements
- increased volumes of data (particularly in data-warehousing
- Many queries lend themselves easily to parallel execution
- Can reduce time required to retrieve relations from disk by partitioning relations onto a set of disks
- Horizontal partitioning usually used. Subsets of a relation are sent to different disks

Round Robin

- Assume n disks.
- With Round Robin: Relation is scanned in some order. The i^{th} relation is sent to disk $D_i mod n$
- Guarantees an even distribution.

Hash Partitioning

- choose attributes to act as partitioning attributes.
- define a hash function with range $0 \ldots n - 1$
- Each tuple is placed according to the result of the hash function

Range Partitioning

- partitioning attribute is chosen
- Partitioning vector is defined $< v_0, v_1, \ldots v_{n-2} >$
- tuples are placed according to value of partitioning attribute. If t[partitioning attribute] < v_0, place tuple t on disk D_0

Query Types

Common types of queries

1. Scanning entire relation (batch processing)
2. Point-Queries (return all tuples that match some value)
3. Range-Queries (return all tuples with some value in some range)

Comparison of Partitioning techniques

- Round Robin
  - useful for batch processing
  - not very suitable for point or range querying as all disks have to be accessed.

Comparison of Partitioning techniques

- Round Robin
  - useful for batch processing
  - not very suitable for point or range querying as all disks have to be accessed.
- Hash partitioning
  - very useful if point query based on partitioning attribute.
  - usually useful for batch querying is a fair hash function is used
  - poor for range querying

Comparison of Partitioning techniques

- Round Robin
    - useful for batch processing
    - not very suitable for point or range querying as all disks have to be accessed.
- Hash partitioning
    - very useful if point query based on partitioning attribute.
    - usually useful for batch querying is a fair hash function is used
    - poor for range querying
- Range Partitioning
    - Useful for point and range querying
    - Can lead to inefficiency in range querying if many tuples satisfy condition

Inter-query Parallelism

- different transactions run in parallel on different processors
- Transaction throughput is increased
- The times for individual queries remains the same
- easiest form of parallelism to implement

Intra-query parallelism

- Can run a single query in parallel on multiple processors (and disks)
- Can speed up running time of query
- Can achieve parallel execution by parallelising individual components (intra-operation parallelism)
- Can also achieve parallel execution by evaluating portions of the query in parallel (inter-operation parallelism)
- Can also combine both

Parallel Sorting

- **Range-Partitioning Sort**
- Distribute the relation using a range-partitioning strategy on the sort attribute
- Each subset is sorted in parallel. The final merge is not expensive due to the range partitioning strategy chosen

Intra-Operation Parallelism

Parallel External Sort-Merge

- Relation is partitioned.
- Each processor P_i sorts the tuples at D_i
- The sorted runs are then merged in parallel.
- Sorted runs are range-partitioned across a set of processors.
- Each processor performs a merge on the incoming streams
- These sorted runs are then concatenated.

Parallel Join

Wish to compute r $\bowtie$ s

Partitioned Join

- Partition relations across the n processors
- Compute $r_0 \bowtie s_0$ at at processor P_0, $r_1 \bowtie s_1$ at processor P_1 etc.
- can partition relations using hash or range partitioning
- suitable for equi-joins; not suitable for other types.

Parallel Join

Wish to compute r $\bowtie$ s

Partitioned Join

- Partition relations across the n processors
- Compute $r_0 \bowtie s_0$ at at processor P_0, $r_1 \bowtie s_1$ at processor P_1 etc.
- can partition relations using hash or range partitioning
- suitable for equi-joins; not suitable for other types.

Fragment and Replicate

- Wish to calculate $r \bowtie_{x>y} s$
- partition r across the processors
- s is replicated at all processors
- $r_i \bowtie_{x>y} s$ is calculated at all processors

Introduction

Single-User System: At most one user at a time can use the system.

Multiuser System: Many users can access the system concurrently.

Concurrency:

> Interleaved processing: concurrent execution of processes is interleaved in a single CPU
>
> Parallel processing: processes are concurrently executed in multiple CPUs.

Concurrency Control, Recovery Mechanisms

Transactions - states, properties; Schedules

Concurrency Control - problems, approaches (locking, timestamping)

Recovery - problems, recovery mechanisms

Transactions - introduction

A transaction: logical unit of database processing that includes one or more access operations (read - retrieval, write - insert or update, delete).

A transaction (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.

Transaction boundaries: Begin and End transaction.

An application program may contain several transactions separated by the Begin and End transaction boundaries.

Reading involves:

finding address of a disk block that contains the item X

copying that disk block to a buffer

copying item X from buffer to program variable X.

Writing involves:

Find the address of the disk block that contains item X.

Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

Copy item X from the program variable named X into its correct location in the buffer.

Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Sample Transaction

```
read_item(X)
X : = X - N
write_item(X)
```

Concurrency Control

In most DBMS environments, it is desirable to allow many people to access the database at the same time.

Hence, many transactions running at once.

Needed to overcome problems that will arise if we allow unchecked access to the database.

The Lost Update Problem:

T1	T2
`read_item(X);`	
`X := X-N;`	
	`read_item(X);`
	`X := X+M;`
`write_item(X);`	
	`write_item(X);`

This results in the `incorrect' value being stored.

Temporary Update Problem:

```
      T1                              T2

   read_item(X);

   X := X-N;

   write_item(X);

                              read_item(X);

                              X := X+M;
                              write_item(X);

   read_item(Y);

       .

       .

    <CRASH>
```

Recovery mechanism will undo the effect of T1; the value of X will be changed back; T2 has the`incorrect' values

Incorrect Summary Problem

occurs when one transaction is calculating a sum (or some other aggregate function) of a range of values and another transaction is concurrently changing those items.

We need means to prevent these types of problems occurring.

Exercise: Draw a sample schedule that shows the incorrect summary problem.

Recovery

If a transaction is submitted to the DBMS, the system should ensure that either:

> the transaction is completed successfully and it's effect recorded *or*
>
> the transaction fails and has no effect on the database.

Partial execution of a transaction should *not* occur

Transactions can fail for a variety of reasons:

System Crash

Transaction Error

Exception Conditions

Concurrency Control Enforcement

Disk Error

Catastrophes

Main operations of a transaction

begin_transaction

read_item or write_item

end_transaction

commit

rollback (a transaction)

Undo (an operation)

Redo (an operation)

States of a transaction

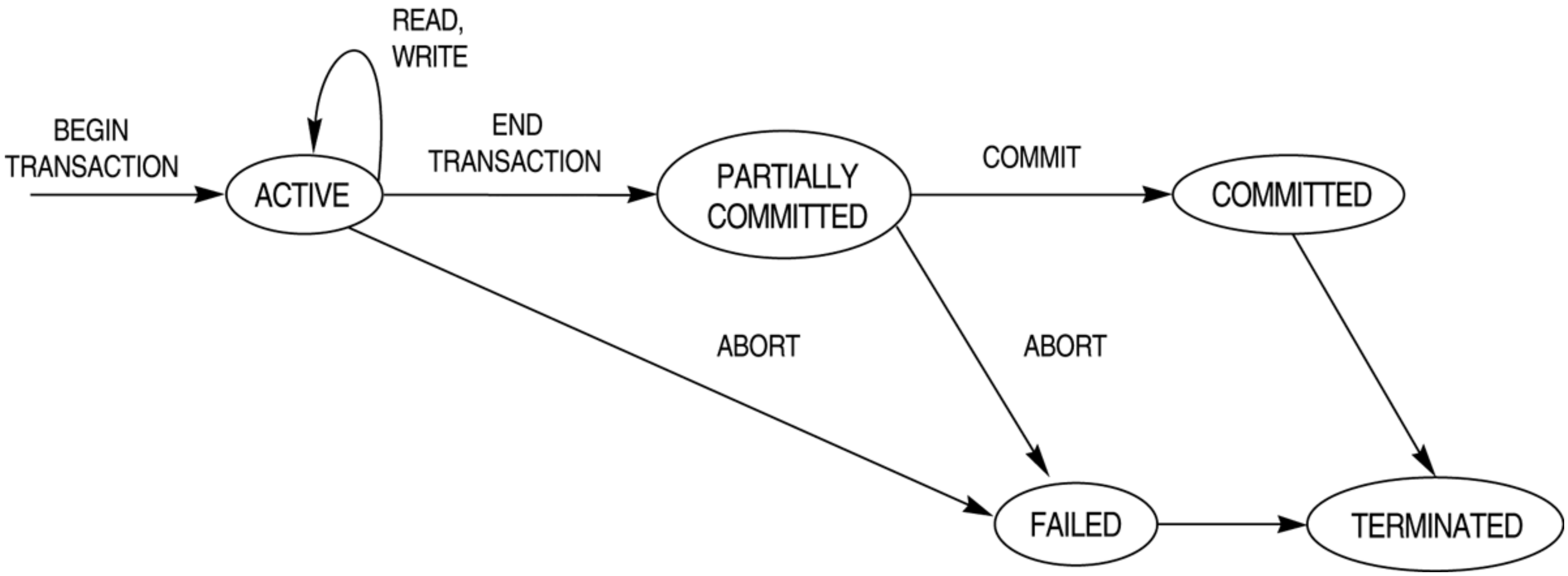
Active state

Partially committed state

Committed state

Failed state

Terminated State


READ,
WRITE
BEGIN
TRANSACTION
ACTIVE
END
TRANSACTION
PARTIALLY
COMMITTED
COMMIT
COMMITTED
ABORT
ABORT
FAILED
TERMINATED

System Log

A *system log or journal* is usually maintained by the DBMS in order to facilitate recovery

The following operations for each transaction are recorded.

System Log

```
start_transaction, T

write_item, T, X, old_value, new_value

read_item, T, X

commit, T
```

Commit Point

A transaction reaches its commit point if:

It finishes successfully

effects are recorded in log

Following a commit point of a transaction any updates by that transaction are considered to permanently stored in the database

A *{commit, T}* entry is recorded in the log

Desirable Properties of transactions:

Atomicity: a transaction should be performed completely or not at all

Consistency Preservation: a transaction should take the database from one consistent state to another

Isolation: updates of a transaction T should not be visible to other transactions until T commits.

Durability: updates made by a committed transaction should not be undone later due to failure.

These four properties are often referred to the ACID properties of a transaction.

Serializability

A schedule is any collection of transactions T_1, T_2, … , T_N). Each transaction can contain a number of read and write operations.

A desirable property of a schedule is that it is serializable.

A serial schedule is a schedule such that there is no interleaving of the operations of the transactions

If a schedule is serial we can guarantee that no lost updates, incorrect summary problems etc. will arise

One potential means to enforce concurrency control is to allow only serial schedules

However, this is far too limiting a constraint and would severely limit the throughput of the system

Ideally, we wish to allow interleaving of operations but maintain `equivalence' to a serial schedule.

A schedule that is `equivalent' to a serial schedule is known as a *serializable schedule.*

Need to define `equivalence' of schedules. The most commonly adopted definition is that of *conflict equivalence*.

Defn: Conflicting operations: 2 operations are said to conflict if (i) they access the same item and (ii) at least one of these operations is a *write.*

We say there is a conflict between two transactions T_1 and T_2 if they contain operations that conflict with each other.

A schedule S is said to be *conflict serializable* if the conflicting operations occur in exactly the same order as in some serial schedule

Given a schedule S of transactions (T_1, … T_N) we can test for conflict serializability using the following algorithm:

for each transaction create a node

Create an edge between node T1 and T2 if:

i) T1 issues read_item(X) before

T2 issues write_item(X) *or*

ii) T1 issues write_item(X) before

T2 issues read_item(X) *or*

iii) T1 issues write_item(X) before

T2 issues write_item(X)

If a cycle exists => not conflict-serializable

else conflict-serializable

Consider again the schedule we had to illustrate the lost update problem.

Graph contains a cycle, hence not serializable

If a cycle exists => not conflict-serializable

else conflict-serializable

This method of checking for conflict serializability is not practical in real world scenarios, as we do not know:

- which transactions will be run
- which operations they will contain.

We need to develop techniques that will guarantee conflict-serializability. We need to reject any operation that violates the principles of conflict serializability

The two main approaches are:

locking protocols

time-stamping

The two main approaches are to guaranteeing conflict-serializability are:

locking protocols

time-stamping

Locking

A *lock* is a variable associated with a data item in the database is used to signify the status of that variable wrt to possible access to the item.

Binary Locks:

A binary lock may have two states: locked or unlocked. If an item is locked by one transaction it cannot be accessed by another transaction

Transaction must be well-formed.

For a transaction `T` to gain any access to a database item `x`, `T` must first issue a `lock(x)` request.

If this request is successful, `T` can access `X`. Otherwise it must wait.

Having completed accessing item `X`, `T` must issue an `unlock(x)` to free up the database item.

Must not unlock free item, or attempt to lock an item it has already locked.

Consider:

```
T1                              T2

read_item(X);
 X := X + 50;

                         read_item(X)
                         X := X - 50;

write_item(X)
                         write_item(X)
```

By applying a locking procedure to the above, T2's `read_item` is queued until T1 finishes and issues an `unlock(X)`

Causes a different ordering of accesses to the database items.

The binary lock approach is too restrictive for real-world applications.

Shared and Exclusive Locks

The `lock(X)` variable can be in one of three states:

read locked
write locked
unlocked

It an item is write locked, no other transaction can obtain a read or write lock on the database item.

If an item is read locked, no other transaction can attain a write lock.

Hence, we allow multiple readers but only one writer.

Transaction can issue `read_lock(x)`, `write_lock(x)` and `unlock(x)`.

Often augmented with `upgrade(x)` and `downgrade(x)`.

These locking schemes on their own do not guarantee serializability.

Two phase locking

They are used in conjunction with locking protocols to ensure correctness. The most commonly used protocol is the 2-phase locking protocol.

Two phase locking (2PL) states that a transaction cannot issue a lock request once an unlock request has been issued.

Hence, there are 2 phases: the growing phases (where transactions obtain locks) and the shrinking phase where the transactions release locks.

2PL locking guarantees conflict serializability.

Exercise: Apply 2PL to the example of the lost update problem and the temporary update problem.

Exercise: Prove that 2PL guarantees conflict serializability (hint: proof by contradiction).

Consider the following schedule:

```
   T1              T2
write_lock(X)
write_item(X)
                 write_lock(Y)
                 write_item(Y)
write_lock(Y)

/* queued */
                write_lock(X)
                 /* queued */
```

-> Deadlock

We need some means to handle deadlock.

We need to first detect deadlock and then resolve the deadlock.

Usually, the lowest priority transaction is terminated. Other transactions can then continue.

Deadlock detection usually involves the creation of a dependency graph. If a cycle exists in the dependency graph, deadlock exists.

Usually triggered by a transaction that has been queued for a certain period of time.

Two-Phase Locking Techniques:

Different variations exist.

Basic: Transaction locks data items incrementally. This may cause deadlock.

Conservative:

Prevents deadlock by locking all desired data items before transaction begins execution.

Strict: A stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm

LOCK(X):

```
B:  if LOCK (X) = 0          //item is unlocked
    then LOCK (X) ← 1  //lock the item
    else begin
              wait (until lock (X) = 0) and
              the lock manager wakes up the transaction);
    goto B
    end;
```

UNLOCK(X):

```
     LOCK (X) ← 0
    if any transactions are waiting for X,
              wake up a waiting transaction.
```

Shared and Exclusive Locks:

read_lock(X)

```
B: if LOCK (X) = “unlocked” then
        begin LOCK (X) ← “read-locked”;
                no_of_reads (X) ← 1;
        end
    else
         if LOCK (X) ← “read-locked” then
                    no_of_reads (X) ← no_of_reads (X) +1
         else
                begin wait (until LOCK (X) = “unlocked” and
                        the lock manager wakes up the transaction);
                  go to B
                end;
```

write_lock(X):

```
B: if LOCK (X) = “unlocked” then
     begin LOCK (X) ← “write_locked”;
          end
     else
          begin wait (until LOCK (X) = “unlocked” and
                    the lock manager wakes up the transaction);
                    go to B
          end;
```

unlock(X):

```
if LOCK (X) = “write-locked” then
    begin LOCK (X) ← “unlocked”;
         wakes up one of the transactions, if any
    end
    else if LOCK (X) ← “read-locked” then
        begin
            no_of_reads (X) ← no_of_reads (X) -1
            if  no_of_reads (X) = 0 then
            begin
                 LOCK (X) = “unlocked”;
                wake up one of the transactions, if any
            end
        end;
```

Timestamping

A timestamp is a unique identifier created by the DBMS to identify a transaction.

Sequentially assigned by the system

We will use TS(T) to identify transaction T.

we associate timestamps with each database item X:

`read_TS`: the largest timestamp of those transactions that have read `X`

`write_TS`: the largest timestamp of those transactions that have written `X`

We can use these and timestamps on transactions to enforce a `timestamp ordering', which will guarantee serializability.

For every request by a transaction `T` to read an item (`read_item(X)`) or to write an item (`write_item(X)`), must check timestamp of transaction with timestamps of database item

If ordering is violated, operation is rejected

`T` issues a `write_item(X)`

```
if (read_TS(X) > TS(T)) or
   (write_TS(X) > TS(T))
     rollback(T)
else
    allow operation
    write_TS(X) := TS(T)
```

`T` issues a `read_item(X)`:

```
if write_TS(X) > TS(T)
    rollback T
if write_TS(X) =< TS(T)
   allow read
   read_TS(X) := max (TS(T), read_TS(X))
```

Time-stamping can be used to enforce conflict serializable schedules:

Consider:

```
T1                    T2
read_item(X);
 X := X+M;
                        read_item(X);
                               X := X-N;
write_item(X);
                        write_item(X);
```

With time-stamping:

Let `TS(T1)` = 1; Let `TS(T2)` = 2;

```
   T1              T2        r_ts(x) w_ts(x)
read_item(X);              1
 X := X+M;
             read_item(X); 2
                X := X-N;

write_item(X);
/* rollback */
              write_item(X);            2
```

Thomas' write rule

A common modification to the timestamping ordering is as follows:

```
write_item(X):

if (read_TS(X) > TS(T))
     rollback(T)
else if write_TS(X) > TS(T)
    ignore write and allow T to continue
else
    allow operation
    write_TS(X) := TS(T)
```

Effect: no longer guarantees conflict serializability but rejects fewer operations.

One of the disadvantages of time-stamping is that a transaction maybe repeatedly aborted and restarted (cyclic restart problem). Similar to unfairness in a locking scheme

Although 2PL and time-stamping both guarantee conflict-serializable schedules, not all schedules allowed under 2PL will be allowed under time-stamping and vice-versa

Multi-version time-stamping:

Time-stamping can be extended to allow multi-version techniques

For each database item `X`, we maintain a set of versions `X1 ...  XN`

If a read or write request is submitted, the timestamp of the transaction can be checked against those of the items involved in the read or write, and the appropriate version used.

Limits the number of transactions that need to be restarted and hence leads to increased concurrency and increased throughput.

Requires much more storage

Granularity

For both locking and time-stamping we have assumed that all lock and timestamps were maintained for database items. The database item could be a record, a field value, a block, a whole file or the entire database.

The larger the item the lower the degree of concurrency.

The smaller the item, the higher the number of locks (or timestamp values) that have to be maintained.

Although 2PL and time-stamping both guarantee conflict-serializable schedules, not all schedules allowed under 2PL will be allowed under time-stamping and vice-versa

Recovery

Basic approach is to maintain a log. If a crash occurs we can scan log for operations to redo and operations that need to be undone.

Recap: A *commit point* of a transaction indicates that a transaction has completed and its effects are considered to reflected in the database. At commit point, we force-write the system log to disk and then append a *[commit, T]* to the system log.

General approach to recovery following a system crash.

Search log for:

transactions that have not yet reached commit point - the effects of these transactions can be undone.

transactions that have reached commit point- the effects of these transactions can be redone.

The system log is also kept on disk. It is common to keep block in memory until it is full. Hence, a part of the log may be lost is there is a system crash.

Hence, if a transaction T reaches its commit point, force write block of log to disk prior to appending *{commit, T}* to log.

A log may become quite large which can cause the recovery process to be quite slow.

Checkpoints are often used to improve performance.

A checkpoint involves:

suspend all transactions

force write all database pages in memory

append *checkpoint* to log

resume all suspended transactions

Usually issued a regular intervals. The recovery system need not look at transactions that have committed prior to the last checkpoint

Transactions usually operate under one of two protocols:

Deferred update:

updates not made to database until the transaction has committed.

Immediate update:

updates are immediately reflected in database

Deferred Update Protocol

Recovery Protocol

1. Examine system log back as far as the last *[checkpoint]* entry, making two lists: *uncommitted* transactions and *committed* transactions.

2. Ignore all the operations of the uncommitted transactions.

3. Redo all the operations of committed transactions.

Under the deferred update protocol, the system log needs only to contain the following entry types:

i) *[start_transaction, T]*
ii) *[write_item, T, X, new_value]*
iii) *[commit, T]*

Immediate Update Protocol

1. Make two lists: *uncommitted* transactions and *committed* transactions.

2. Undo all the operations of the uncommitted transactions

3. Undo all the operations of committed transactions that have read an item of previously written by a rolled back transaction.

4. Redo all the operations of committed transactions that have not read an item by a rolled back transaction

Can have cascading rollback.

We need to keep extra records in log to facilitate recovery:

[write_item, T, X, old_value, new_value]
[read_item, T, X]

SQL Support for Transactions

A single SQL statement is always considered to be atomic. Either the statement completes execution without error or it fails and leaves the database unchanged.

In SQL, there is no explicit Begin Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered.

Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

Can specify the characteristics of an SQL statement with the SET command.

Can specify the access mode and isolation level.

Access mode: READ ONLY or READ WRITE.

The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.

Isolation level <isolation>, where <isolation> can be:

READ UNCOMMITTED,
READ COMMITTED,
REPEATABLE READ
SERIALIZABLE.

The default is SERIALIZABLE.

With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability. However, if any transaction executes at a lower level, then serializability may be violated.

Potential problem with lower isolation levels:

Temporary Update Problem:

Reading a value that was written by a transaction which failed.

Nonrepeatable Read:

Allowing another transaction to write a new value between multiple reads of one transaction.

Phantoms:

New rows being read using the same read with a condition.

Sample SQL transaction:

```
 EXEC SQL whenever sqlerror go to UNDO;
 EXEC SQL SET TRANSACTION
             READ WRITE
             DIAGNOSTICS SIZE 5
             ISOLATION LEVEL SERIALIZABLE;
 EXEC SQL INSERT
            INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
            VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
            SET SALARY = SALARY * 1.1
            WHERE DNO = 2;
EXEC SQL COMMIT;
            GOTO  THE_END;
 UNDO: EXEC SQL ROLLBACK;
 THE_END;
```

Summary:

Concurrency control – potential problems.

Transaction – states

Desirable properties of transactions

Schedules – serial, serializability

2 phase locking, timestamping

Recovery – system log, entries, algorithms for recovery

More Database Models: distributed databases

November 7, 2023

In a centralised database management system all system components and data reside at a single site
In recent years, there has been a growing trend towards distributed database systems.

- Distributed nature of some database applications: a company may have many different locations and data at each location
- Increased reliability and availability: if data and DBMS s/w is distributed, then user is not dependent on just one site
- Data Sharing
- Improved Performance: local queries may be executed more efficiently.

- Distribution leads to increased complexity in system design and implementation:
- Additional functionality is required:
  - to allow access to remote sites
  - to keep track of data distribution data and replication
  - to devise execution strategies for queries and transactions that access many sites
  - to maintain consistency across sites
  - to allow recovery from new types of failures

Data Fragmentation

Horizontal Fragmentation

Fragment tables into sub-tables based on certain SELECT restrictions.

Vertical Fragmentation

Sets of attributes from a table are stored at different sites. This type of fragmentation can be defined by a PROJECT operation.
Vertical fragmentation is never totally disjoint as the key attributes must be stored at each site. Necessary in order to reconstruct the table.

Hybrid fragmentation

Defined by a sequence of SELECTs and PROJECTs

Fragmentation Schema

A fragmentation schema (which will be in every data catalog for each client) is a full set of fragmentation definitions.

Allocation Schema

An allocation schema (also in the catalog) defines the location of fragments

Replication

- Useful in improving availability of data
- Full Replication: store whole database at each site
- Partial Replication: replicate certain fragments
- No replication

Query processing

- In a centralised DBMS, we attempt to maximise efficiency by reducing the size of intermediate tables.
- In distributed DBMS, the most significant measure of cost is the quantity of data transferred.
- The most common execution strategies are based on reducing network traffic.
- The $semi - join$ “operator”, is the standard approach where no redundant tuples or attributes are transferred. Only attributes needed in join conditions or in the final result are transferred.

Distributed Query Processing: small example

- Assume relations *employee*, *dept*, and *project* are stored at $site_1$, $site_2$ and $site_3$ respectively.
- Assume also no fragmentation or replication of these relations.
- We wish to join tables to obtain the result at some site $site_i$, i.e we need to compute employee $\bowtie$ dept $\bowtie$ project.

- No one strategy is always the best.
- The relations involved, their size, selectivity of joins etc. will all vary over time.

Distributed Query Processing: semi-join

- The semi-join operator is a commonly adopted approach to guarantee some degree of efficiency.
  Let relations r and s be at $site_1$ and $site_2$ respectively. We wish to calculate $r \bowtie s$.
  Often, there will be many tuples in r and s that will not be included in the result.

Distributed Query Processing: semi-join

1. Create *tmp*1 comprising the join attributes of *r*
2. Ship *tmp*1 to $site_2$
3. Execute $s \bowtie tmp1 : -tmp2$ at $site_2$
4. Ship *tmp*2 to $site_1$
5. Evaluate $r \bowtie tmp2$ at $site_1$

Usually reduces the number of spurious tuples to be transferred.

Concurrency Control and Recovery

Concurrency Control and Recovery

Numerous problems arise in distributed databases that do not exist in a centralised DBMS:

- Dealing with multiple copies of data items. The concurrency control mechanism must ensure consistency between these items
- Failure of individual sites: The DBMS should continue to operate; and when the site recovers it should be brought up to date
- Failure of communication links
- Distributed Commit
- Distributed Deadlock

Recovery

- In order to facilitate recovery we must generate atomicity of transactions.
- This becomes a more difficult problem in distributed databases as a transaction must commit at all sites or must fail at all sites.
- A two-phase commit procedure is usually adopted.
- A transaction coordinator is located at one site, $site_i$
- When a transaction T completes execution coordinator is informed.

Phase 1

- [*prepare*, *T*] is added to log at *site*$_i$; log is force-written.
- [*prepare*, *T*] message is sent by the coordinator to all involved sites.
- Transaction managers at sites return an [*abort*, *T*] message or a [*ready*, *T*] message (whether T has successfully terminated or not).
- If [*ready*, *T*] entry is sent by a site, individual logs are then force-written.

Phase 2

- if all sites respond with a [*ready*, *T*] (within in given time), the transaction is considered committed.
- [*commit*, *T*] is added to the log. Force-write log.
- else, [*abort*, *T*] is placed. Force-write log.
- Coordinator then informs all sites as to whether T has committed or not.
- Variations on this approach. Most of these variations attempt to increase the efficiency of recovery.

- These algorithms require a coordinator. The coordinator is usually chosen in advance. Measures have to be taken to ensure correctness if the coordinator happens to crash.
- Backup Coordinator: maintains up-to-date copy of coordinator. Can be extended to have a chain of backups.
- Election protocols: If the coordinator crashes, any involved sites may try to assume control. If they obtain the majority of votes, they assume control and inform all others.

Concurrency Control

- Most approaches merely extend centralised approaches of 2-phase locking and time-stamping:
- With locking a single Lock Manager can be chosen: one site chosen as lock manager. All locks are granted by the lock manager at this site.
- Advantage: Easy to implement.
- Disadvantages: Leads to bottleneck at lock manager site; particularly if fine-grained locking used.
- Over-dependence on one site

Multiple lock managers

- Each site possesses its own lock manager for items present at that site.
- For non-replicated items, no real problems arise.
- For items replicated at many sites, a transaction issuing a *write_item* needs to send a request to all lock managers. Each lock manager sends an acceptance or a rejection.
- A majority protocol is typically used, i.e., if the majority of responses are grants, then transaction obtains lock.

Distributed Deadlock

- One potential problem of distributed locking protocols is the possibility of distributed deadlock.
- Further complicated by the potential of phantom deadlocks.
- Many algorithms exist to try and efficiently deal with this problem.

Timestamping

- Time-stamping can also be extended:
- Timestamps generated at each local site
- Difficulties arise with respect to ordering transactions. If some sites have higher throughput, they will have higher timestamps and hence timestamp ordering will be invalid.
- Usually create timestamp by actualy taking combining actual timestamp and site identifier.
- Ordering is usually enforced by using logical clock schemes.

Logic Databases
(deductive databases)

Incorporates ideas from logic and artificial intelligence.

In addition to storing data, can store rules to infer or deduce new facts.

A declarative language to specify facts and rules.
A variation of Prolog is used (Datalog).

An inference engine is provided to deduce new facts.

Facts are similar to instances of relations in relational databases.

In a RDBMS, the interpretation is suggested by the attribute names; in deductive databases, the interpretation an value depends on its position.

Rules can also be specified. Emphasis in deductive databases has been in the inference of new
facts.

Prolog

A predicate: contains a name and a set of arguments.

If the arguments are constant values (literals) then we are specifying a fact.

If some or all of the arguments are variables then we are specifying a rule.

Constants: denoted by using a lower case letter as the first letter.

```
supervise (jim, john).
supervise (john, jack).
supervise (jack, joe).

superior(X,Y) :- supervise(X,Y).
superior(X,Y) :- supervise(X,Z), superior(Z,Y).
```

Queries can be satisfied by checking for matches in the set of facts, or by applying rules to facts to infer new facts and comparing these to the query goal.

Two resolution techniques are possible:

- Backward chaining (aka Top-down): start with query goal and attempt to match with facts and rules (unification). If solving a goal with a set of subgoals, satisfy in a left-to-right manner.

Pred :- Pred1, Pred2, Pred3, Pred4.

- Forward chaining (aka Bottom-Up): check if facts match query. Then apply each rule to all facts and rules, inferring a set of facts; each inferred fact if compared to the query predicate.

Backward chaining is far more efficient. A large set (potentially infinite) of spurious facts can be generated by the forward chaining techniques.

Prolog systems and deductive databases adopt a backward chaining approach.

The deductive database is queried by specifying a predicate goal.

If the goal contains literals only, then a Boolean value is returned.

If the goal contains variables, then a set of facts are returned that render the query goal true

Safety of Programs

A program (or rule) is said to be safe if it returns a finite set of facts.
To determine if a set of rules is safe or not is undecidable.

Consider the employee schema we met earlier.

We wish to define a rule that returns employees with a large salary

```
large_salary(Y) :- Y > 100000,
                   employee(X),
                   salary (X, Y).
```

Assume we have facts:

```
employee(jim).
employee(jack).
salary(jim, 30000).
salary(jack, 110000).
```

Consider:

```
big_salary(Y) :- employee(X),
                 salary (X, Y),
                 Y > 100000.
```

Can also run into problems with recursive rules and ordering of rules and facts.

Consider rules:

```
...
pred1(X) :- pred(Y).
pred(Y) :- pred1(X).
...
```

Consider rules:

```
fact (N, X) :-  N1 is N -1,
                fact (N1, X1),
                X is X1 * N.

fact (0, 1).
```

It is clearly important to guarantee safety of programs.

A rule is safe if it generates a finite number of facts.

More formally, a rule is safe if all variables are limited.

A variable X is limited if:

1. it appears in a regular predicate in the rule body

2. it appears in a predicate of form X=c or (c1 <= X and X <= c2) where c, c1 and c2 are constants

3. it appears in a predicate X=Y or Y=X, where Y is a limited variable

Relational Operators

Can easily specify relational operators as Datalog rules.

Allows incorporation of relational views and queries

Example

Assume we have 3 relations each with three arguments
rel_one(A,B,C).
rel_two(D,E,F).
rel_three(G,H,I,J).

Can specify select queries as follows:

```
select_one_A_eq_c(X, Y, Z) :- rel_one(c, Y, Z).

select_one_A_eq_c_and_B_less_5(X, Y, Z) :- rel_one(c, Y, Z),
                                  Y < 5.

select_one_A_eq_c_or_B_less_5(X, Y, Z) :-  rel_one(c, Y, Z).

select_one_A_eq_c_or_B_less_5(X, Y, Z) :-   Y < 5.
```

Can specify project as follows:

```
project_two_D_F(X,Z) :- rel_three(X,Y,Z).
```

Set operators:

```
union_one_two(X,Y,Z) :- rel_one(X,Y,Z).
union_one_two(X,Y,Z) :- rel_two(X,Y,Z).
```

Note that if rel one and rel two contain matching tuples (facts), we will have duplicates in the result. This isn’t strictly correct. Can rewrite as:

```
union_one_two(X,Y,Z) :- rel_one(X,Y,Z).
union_one_two(X,Y,Z) :- rel_two(X,Y,Z),
not(rel_one(X,Y,Z)).
```

```
intersect_one_two(X,Y,Z) :- rel_one(X,Y,Z),
rel_two(X,Y,Z).

difference_one_two(X,Y,Z) :- rel_one(X,Y,Z),
                             not(rel_two(X,Y,Z)).


cartesian_one_three(T,U,V,W,X,Y,Z) :-         rel_one(T,U,V),
                                              rel_three(W,X,Y,Z).
```

Hybrid operators (join):

```
join_one_three_C_eq_G(U,V,W,X,Y,Z):-
                                    rel_one(U,V,W),
                                    rel_three(W,X,Y,Z).
```

Security in Databases

November 14, 2023

Issues

- Legal and Ethical Issues
- Policy Issues
- System Issues - levels at which security should be enforced.
- Security Levels

- DBMS typically includes security and an authorisation systems.
- Areas of consideration:
  - Preventing unauthorised access
  - Access systems
    1. discretionary
    2. mandatory
  - Statistical database security.

The database administrator (DBA) has access to a number of commands for granting and revoking access for users and groups. These include:

- account creation
- privilege granting
- privilege revocation
- security level assignment

Access Protection

- All users have a user name and password.
- Keep track of all operations (particularly updates)
- expand system log.

Operations against the database may be controlled.
Two levels of assigning privileges:

- account level
- relation level

Account level

Capabilities provided for the account:
These include CREATE SCHEMA, CREATE VIEW, ALTER, DROP, MODIFY, SELECT

Access rights provided for a relation

- follows the access matrix model
- rows correspond to subjects
- columns correspond to objects
- Mi, j corresponds to the privilege subject i has on object j
- Privilege $\in$ {read, write, update}

Can be extended in SQL to allow the following privileges:

- SELECT
- MODIFY (UPDATE, DELETE, INSERT)
- REFERENCES (can refer to relation R, when specifying referential integrity)

- Can specify privileges using VIEWS.
- Create a view over a base relation (or set of).
- Define privileges on R.

Propagation of Privileges

One can grant privileges with the GRANT option.

```
GRANT SELECT
ON EMPLOYEE
TO user22
WITH GRANT OPTION
```

Limiting Propagation

One can grant privileges with the GRANT option.
Techniques exist based on horizontal and vertical limits.

- Horizontal: can grant to at most i users
- Vertical: limits ‘depth’ of granting grants. Vertical limit zero is equivalent to granting privilege without the grant option.

- Allows a number of security classes (e.g. TOP SECRET, SECRET, CLASSIFIED, UNCLASSIFIED)
- Can be used with discretionary access control.
- Can have a number of security classes that form a lattice.
- Classify subjects as belonging to a class.
- Classify objects as belonging to a class.

Two restrictions/Properties (Bell-LaPadula Model

- A subject S is not allowed read access to an object O unless: $class(S) \geq class(O)$ (simple security property)
- A subject S is not allowed to write to an object O unless: $class(S) \leq class(O)$ (star property)

In order to incorporate multi-level security notions, we can associate classification attributes with every attribute.

- The schema then becomes

$$R(A_1; C_1; A_2; C_2; \ldots A_n; C_n; TC)$$

  where TC is the classification of the tuple, set to be $max(C_1, ..., C_N)$

Apparent Key

The apparent key is the set of attributes that would ordinarily form the key.

- store entire tuple at a high classification and produce lower-level classications through 'filtering'
- polyinstantiation: multiple copies of the same tuple. Also requires modified definitions with respect to integrity rules.

Statistical Databases

- Used to produce statistics on various ‘populations’
- Individual tuples are classified.
- Queries involve applying statistical functions to a population of tuples.
- Only allow: COUNT, SUM, MIN, MAX, AVERAGE. STANDARD DEVIATION.
- Still potential may exist for ‘inference’ of classified data.

- Q1: SELECT COUNT(*) FROM <relation> WHERE <condition>
- Q2: SELECT AVG(<attribute> FROM <relation> WHERE <condition>

By modifying <condition>, we can infer data.

Can use this idea to create ‘linear set of equations’:

- `Query 1 = cond1 AND cond2 AND cond3`
- `Query 2 = cond2 AND cond3`
- `Query 3 = cond1 AND cond3`

Prevention Techniques

- Apply to query - track user queries and disallow query in the sequence that infers data. Very difficult to do.
- Apply to data
  - Suppression
  - Concealment/Disguise