BACHELOR OF SCIENCE IN COMPUTER SCIENCE

YEAR 4 SEMESTER 2

CCS 418: ADVANCED DATABASE SYSTEMS

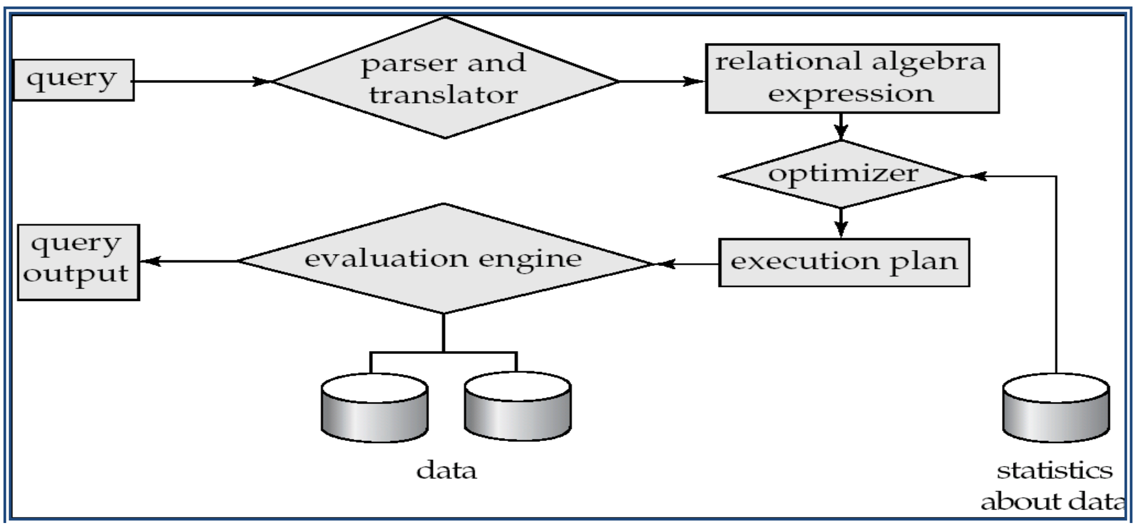
NOTES 2

QUERY PROCESSING

- Overview
- Measures of ***Query*** **Cost**
- ***Selection*** Operation
- ***Sorting***
- ***Join*** Operation
- *Other* Operations
- Evaluation of ***Expressions***

Basic *Steps* in *Query Processing*

1. *Parsing* and translation
2. Optimization
3. Evaluation



- ***Parsing*** and **translation**
  - ***translate*** the *query* into its ***internal form***.
    - This is then *translated* into *relational* ***algebra***.
  - ***Parser*** checks *syntax*,
    - verifies relations
- **Evaluation**
  - The query-execution engine takes a ***query-evaluation plan***,
    - ***executes*** that *plan*, and
    - ***returns*** the *answers* to the query.
- **Query Optimization**: Amongst ***all*** *equivalent* ***evaluation plans***:

        - ***choose*** the **one** *with* ***lowest cost***.
        - **Cost** is ***estimated*** using ***statistical information*** from:
            - the *database catalog*
            - *e.g. number of tuples* in each relation, *size of tuples*, etc.
- In *this chapter* we study:
    - ***How*** to *measure* ***query costs***
    - ***Algorithms*** for ***evaluating*** *relational algebra* **operations**
    - **How** to ***combine algorithms*** for *individual operations* in order to:
        - *evaluate* a *complete expression*

***Measures* of *Query* Cost**

- **Cost** is generally ***measured*** as:
        - **total elapsed time** for *answering query*
    - *Many* ***factors*** *contribute* to **time cost:**
        - *disk accesses, CPU*, or even network *communication*
- Typically ***disk access*** is the *predominant* cost, and
        - is also *relatively* ***easy*** *to* ***estimate***.
    - *Measured* by taking into account:
        - *Number of seeks* ***** *average-seek-cost*
        - *Number of blocks read* ***** *average-block-read-cost*
        - *Number of blocks written* ***** *average-block-write-cost*
        - ***Cost*** to ***write*** a *block* is ***greater*** than *cost* to ***read*** a *block*
            - data is *read back* after being *written*:
                - » to *ensure* that the *write* was *successful*
- As the ***cost*** *measures*:
    - for *simplicity* we just use:
        - the *number of* ***block transfers*** *from disk and*
        - *the number of* ***seeks***
    - t_T – *time* to *transfer one block*
    - t_S – *time* for *one seek*
    - **Cost** for **b** *block transfers* plus **S** *seeks:*

        $Cost = b * t_T + S * t_S$
- We **ignore** *CPU costs* for *simplicity*
    - *Real systems* do ***take*** *CPU cost into account*
- We do **not** ***include*** *cost* to ***writing output*** to *disk* in our cost formulae
- *Several* ***algorithms*** can ***reduce disk*** **I/O** by:
        - using *extra buffer space*
    - *Amount* of ***real memory*** *available* to ***buffer*** *depends on*:
        - other *concurrent* ***queries*** and **OS** *processes*,
            - *known* ***only*** *during execution*
        - We often use ***worst*** *case* ***estimates***, assuming:
            - *only* the ***minimum*** amount of ***memory***
                - needed for the operation is ***available***
- ***Required data*** *may* be ***buffer resident*** already,
        - *avoiding* ***disk I/O***
    - But ***hard*** to *take into account* for ***cost estimation***

Selection Operation

- **File scan** *algorithms*:
  - search *algorithms* that ***locate*** and ***retrieve*** *records*
    - that ***fulfill*** a *selection condition*. (***No*** *Index use***!**)
- Algorithm **A1** (***linear*** *search*): ***Scan*** each ***file block*** and
  - *test* all *records* to see whether they ***satisfy*** the *selection* ***condition***.
  - ***Cost*** estimate = $\boldsymbol{b_r}$ *block transfers* + **1** *seek*
    - $\boldsymbol{b_r}$ denotes *number of* ***blocks*** containing records from ***relation r***
  - **If** *selection* is on a ***key*** *attribute*:
    - *can* ***stop*** on *finding* record
    - cost = ($\boldsymbol{b_r}$ **/2**) *block transfers* + **1** *seek*
  - ***Linear*** *search* can be applied ***regardless*** *of* :
    - *selection* ***condition*** or
    - ***ordering*** *of records* in the file, or
    - *availability of* ***indices***
- **A2** *(**binary** search).* Applicable if *selection* is:
  - an ***equality*** *comparison* :
  - *on* the ***attribute*** *on which* ***file*** is ***ordered***.
  - *Assume* that the ***blocks*** of a relation are ***stored contiguously***
  - ***Cost*** *estimate* (*number of* ***disk blocks*** to be ***scanned***):
    - *cost* of ***locating*** the ***first*** *tuple* by a *binary search* on the blocks
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - **If** there are *multiple records* satisfying *selection*
      - *Add transfer cost of the number of* **blocks** containing records that ***satisfy*** *selection condition*

Selections Using *Indices*

- **Index scan** *algorithms*: search *algorithms* that ***use*** an *index*
  - So, *selection condition must* be on *search-key* of an *index*.
- **A3** (*primary index on* ***candidate key***, *equality*).
  - *Retrieve* a ***single*** *record* that *satisfies* the corresponding *equality condition*
    - *Cost* = $(h_i + 1) * (t_T + t_S)$
- **A4** (*primary index on* ***nonkey***, *equality) :*
  - *Retrieve* ***multiple*** *records*.
  - *Records* will be on *consecutive blocks*
    - Let **b** = *number of* ***blocks*** containing *matching records*
    - *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$
- **A5** (*equality on search-key of secondary index).*
  - *Retrieve* a ***single*** *record* **if** the *search-key* is a *candidate key*
    - *Cost* = $(h_i + 1) * (t_T + t_S)$
  - Retrieve ***multiple*** *records* **if** *search-key* is **not** a *candidate key*
    - *each* of $\boldsymbol{n}$ *matching records* may be on a *different block*
    - Cost = $(h_i + n) * (t_T + t_S)$

    be *very expensive***!**

Selections Involving *Comparisons*

- Can ***implement*** *selections* of the *form* $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$ by *using*:
  - a ***linear*** *file* ***scan*** or ***binary*** *search*,
  - ***or*** by *using* ***indices*** in the *following ways*:
- **A6** (*primary index, comparison).* (*Relation* is ***sorted*** on **A**)
  - For $\sigma_{A \ge V}(r)$ *use* ***index*** to *find first tuple* $\ge v$ and
    - *then scan* relation *sequentially from there*
  - For $\sigma_{A \le V}(r)$ *just scan* relation *sequentially till first tuple* > v;
    - do **not** *use index*
- **A7** (*secondary index, comparison*).
  - For $\sigma_{A \ge V}(r)$ *use index* to *find first index entry* $\ge v$ and
    - then *scan* index *sequentially* from there,
      - to *find pointers* to *records*.
  - For $\sigma_{A \le V}(r)$ *just scan* ***leaf pages*** of *index finding pointers* to *records*,
    - *till first entry* > v
  - In *either case, retrieve records* that are *pointed to*,
    - *requires* ***an I/O*** for *each record* **!**
    - *Linear file scan may* be *cheaper* **!**

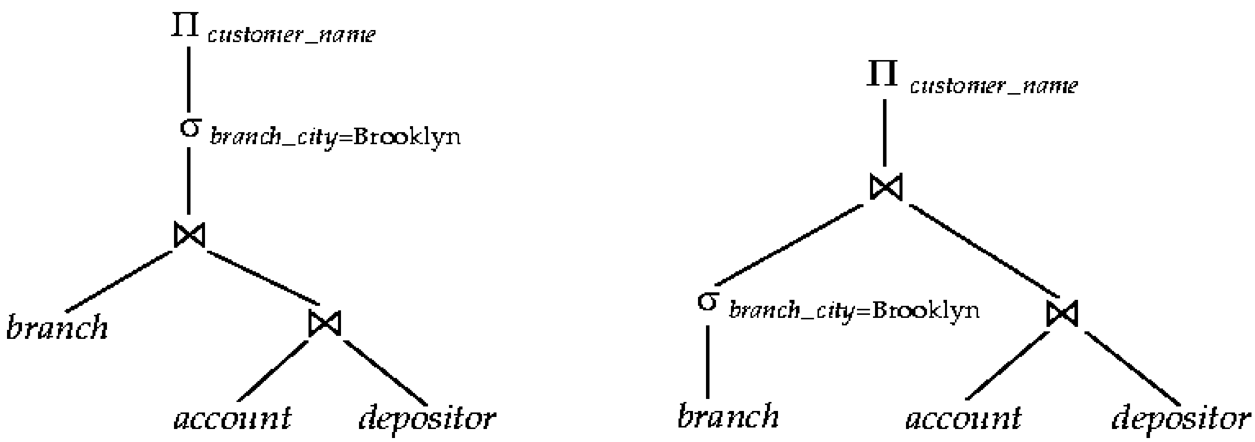
QUERY OPTIMIZATION

Introduction

- ***Alternative ways*** of *evaluating* a given *query*
  - *Equivalent expressions*
  - *Different algorithms* for each operation
- ***Cost difference*** between a ***good*** and a ***bad way*** of *evaluating a query*
  - *can* be ***enormous!***
- ***Need*** to ***estimate*** the ***cost*** *of operations*
  - ***Statistical*** *information about relations*.

  E.g.:
    - *number of tuples*,
    - *number* of *distinct values* for an attributes,

  - Statistics ***estimation*** for ***intermediate*** *results*
    - to ***compute cost*** of ***complex*** *expressions*
- *Relations generated* **by** two ***equivalent expressions*** :
  - have the **same** *set of attributes* and
  - contain the **same** *set of tuples*
  - *although* their tuples/attributes *may be* ***ordered differently***.



(a) Initial expression tree (b) Transformed expression tree

- ***Generation*** of *query-evaluation* ***plans*** for an expression
  - involves ***several*** *steps*:
  1. *Generating* logically ***equivalent*** *expressions* using *equivalence* ***rules***.
  2. *Annotating* resultant ***expressions*** to *get alternative query plans*
  3. *Choosing* the ***cheapest plan*** based on *estimated cost*
- The overall process is *called* ***cost based optimization*.**

***Transformation* of Relational Expressions**

- **Two** relational algebra **expressions** are *said to be* **equivalent:**
  - **if** on every legal *database instance* the ***two*** *expressions* generate
    - the **same** ***set*** *of tuples*
  - *Note*: **order** of tuples ***is*** *irrelevant!*
- In SQL, inputs and outputs **are** ***multisets*** *of tuples***:**
  - **Two** ***expressions*** in the multiset version of the relational algebra are *said to be* equivalent**:**
    - **if** on every legal *database instance* the ***two*** expressions generate
      - the **same** ***multiset*** *of tuples*
- An **equivalence rule** *says* that:
  - **If** *expressions* of ***two*** *forms* are equivalent**:**
    - Can **replace** expression of **first** form by **second**,
    - **or** *vice versa*

Enumeration of *Equivalent* Expressions

- *Query optimizers* use *equivalence rules* to systematically
  - *generate expressions equivalent* to the given expression
- Conceptually, generate *all equivalent* expressions by repeatedly executing the following step until no more expressions can be found:
  - for each *expression* found so far,
    - use all applicable *equivalence rules*
    - add *newly generated* expressions to the *set of expressions* found so far

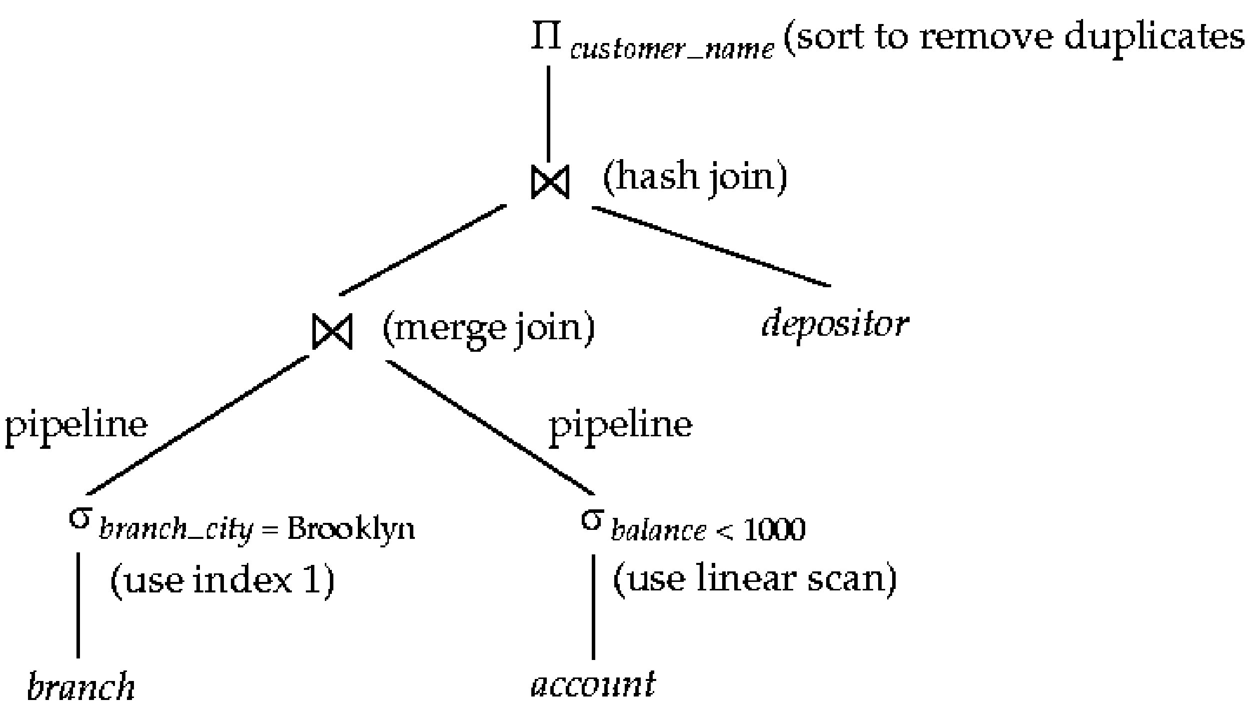
- The above approach is *very expensive* in *space* and *time*
- *Space requirements* reduced by *sharing* common *subexpressions*:
  - when E1 is *generated* from E2 by an *equivalence* rule:
    - usually *only* the *top level* of the two *are different*,
    - *subtrees* below are the *same* and *can be shared*
    - *E.g*. when applying *join associativity*
- *Time requirements* are reduced by *not generating all* expressions

Cost **Estimation**

- Cost of *each operator computes*
  - Need *statistics* of *input relations*
    - *E.g. number of tuples, sizes of tuples*
- *Inputs can be results* of *sub-expressions*
  - Need to *estimate statistics* of expression *results*
  - To do so, we *require additional statistics*
    - *E.g. number of distinct values* for an *attribute*

Evaluation **Plan**

- An *evaluation plan* defines exactly
  - *what algorithm* is used for *each* operation, and
  - *how* the *execution* of the operations is *coordinated*.



Choice **of Evaluation Plans**

- Must *consider* the interaction of *evaluation techniques*

- when choosing evaluation plans:
  - choosing the cheapest algorithm for *each* operation independently
    - may not *yield best overall algorithm*.
- E.g., merge-join may be costlier than hash-join,
  - but may provide a *sorted output*
    - which *reduces* the cost for an *outer level aggregation*.
- nested-loop join may *provide opportunity* for pipelining

- Practical query optimizers incorporate elements of:
  - the following two broad approaches:

1. Search *all the plans* and choose the *best* plan in a *cost-based* fashion.
2. Use *heuristics* to choose a plan.

Cost-Based **Optimization**

- Consider finding the best *join-order* for $r_1 \bowtie r_2 \ldots r_n$.
- There are $(2(n-1))! / (n-1)!$ different *join orders* for above expression.
  - With $n = 7$, the number is 665280,
  - with $n = 10$, the number is greater than 17.6 billion!

BUT:

- No need to generate all the *join orders*.
- Using *dynamic programming*:
  - the least-cost *join order* for any subset of $\{r_1, r_2, \ldots r_n\}$ is:
    - *computed* only once and
    - *stored* for *future use*.

Dynamic Programming **in Optimization**

- To find best join tree for a set of n relations:
  - To find best plan for a set S of n relations,
    - consider all possible plans of the form: $S_1 \bowtie (S - S_1)$
      - where S_1 is any *non-empty* subset of S.
        - » i.e., $2^n - 1$ *alternatives!*
        - » *e.g., 1023 for* $\{r_1, r_2, \ldots r_{10}\}$
  - Recursively,
    - *Compute* costs for joining subsets of S to find the cost of each plan.
    - *Choose* the cheapest of the $2^n - 1$ *alternatives*.
  - When a plan for any subset is *computed*,
    - *store* it and *reuse* it when it is required again,
      - instead of recomputing it.

Heuristic Optimization

- Cost-based optimization is expensive,
  - **even with** dynamic programming.
- **Systems** may **use *heuristics*** to **reduce**
  - the number of choices that must be made in a cost-based fashion.
- **Heuristic** optimization **transforms** the **query-tree** by

    - using a set of rules that typically (but *not in **all** cases*)
    - **improve** execution performance:
  - Perform *selection early* (reduces the number of tuples)
  - Perform *projection early* (reduces the number of attributes)
  - Perform most restrictive selection and join operations
    - **before** other similar operations.
  - **Some** systems use only heuristics,
    - **others** combine heuristics with *partial cost-based optimization*.

Structure of Query Optimizers

- The **System R**/Starburst **optimizer** considers **only** left-deep join orders.
  - This reduces optimization complexity and
  - generates **plans** amenable to pipelined evaluation.
  - **System R**/Starburst also
    - uses **heuristics** to **push** selections and projections **down** the query tree.
- Heuristic optimization used in some versions of **Oracle**:
  - Repeatedly pick "best" relation to join next
    - Starting from each of **n starting points**.
    - Pick best among these.
- For scans using secondary indices,
  - **some optimizers** take into account the **probability** that
    - the page containing the tuple is in the buffer.
- Intricacies of SQL complicate query optimization
  - E.g. **nested** subqueries

- Some query optimizers integrate
    - heuristic selection and
    - the **generation** of alternative ***access plans***.
  - **System R** and **Starburst** use a *hierarchical procedure* based on
    - the ***nested-block*** concept of SQL:
      - *heuristic* rewriting followed by cost-based *join-order optimization*.
- **Even with** the *use of **heuristics***,
  - *cost-based **query optimization*** **imposes** a *substantial* **overhead**.
- This **expense** is usually ***more than offset*** by:
  - **savings** at query-execution time,
    - ***particularly*** by ***reducing*** the number of ***slow disk*** accesses.

Optimizing *Nested Subqueries*

- **SQL** conceptually **treats** nested subqueries in the ***where*** clause
  - **as** functions that take ***parameters*** and
    - **return** a single value or set of values
  - ***Parameters*** are *variables from **outer level*** query that are
    - used in the nested subquery;
    - such variables are ***called*** **correlation variables**

- *E.g.* **select** *customer_name*
  from *borrower*
  where exists (**select** *
  from *depositor*
  where *depositor.customer_name* =
  borrower.customer_name)
- Conceptually, nested subquery is **executed** *once for each tuple*
  - in the *cross-product* generated by the *outer level* **from** clause
  - Such evaluation is called **correlated evaluation**
  - Note: other conditions in **where** clause may be
    - **used** to compute a **join** (instead of a *cross-product*)
    - ***before*** executing the *nested subquery*

- Correlated evaluation **may** be **quite inefficient** since:
  - a *large number of calls* may be made to the *nested subquery*
  - there may be unnecessary **random I/O** as a result
- **SQL *optimizers*** attempt to
  - **transform** nested subqueries to **joins** where *possible*,
  - **enabling** use of *efficient* ***join techniques***
    - *E.g.*: earlier nested query can be rewritten as:

select *customer_name*
from *borrower, depositor*
where *depositor.customer_name = borrower.customer_name*

- Note: above query *doesn't* ***correctly*** *deal with duplicates*, can be *modified* to do so as we will see
- ***In general***, it is **not *possible*** / ***straightforward***
  - to **move** the **entire** nested subquery into the *outer level query*
  - A temporary relation is ***created*** instead, and
    - used in ***body*** of *outer level query*

In general, SQL queries of the form below can be rewritten as shown

- Rewrite: **select** …
  from L_1
  where P_1 **and exists** (**select** *
  from L_2
  where P_2)
- To: **create table** t_1 as
  select distinct V
  from L_2
  where P_2^1
  select …
  from L_1, t_1
  where P_1 **and** P_2^2
  - P_2^1 contains *predicates* in P_2 that do *not involve* any *correlation variables*
  - P_2^2 reintroduces *predicates involving correlation variables*, with relations renamed appropriately
  - **V** contains ***all attributes*** used in *predicates* with *correlation variables*

- *In our example,* the original nested query would be **transformed** to:

create table t_1 **as**
select distinct *customer_name*
from *depositor ;*
select *customer_name*
from *borrower*, t_1
where t_1*.customer_name = borrower.customer_name*

- The process of replacing a nested query **by** a *query with a join*
    - (possibly with a *temporary relation*)
  - is *called* **decorrelation**.
- **Decorrelation** is more **complicated** when
  - the nested subquery ***uses*** aggregation, or
  - when ***the result*** of the nested subquery is ***used*** to test for equality, or
  - when the condition *linking* the nested subquery *to the other query*
    - is **not exists**,
  - and *so on*.

Materialized Views

- A **materialized view** is
  - a **view** *whose* contents are computed and stored.
- Consider the view:

create view ***branch_total_loan***(*branch_name*, *total_loan*) **as**
select *branch_name*, **sum**(*amount*)
from *loan*
groupby *branch_name*

  - Materializing the above **view** would be **very useful**
  - if the **total loan amount** is **required** frequently
  - **Saves** the **effort** of
    - *finding multiple tuples* and
    - *adding up* their amounts

Materialized View *Maintenance*

- The task of **keeping** a *materialized view* **up-to-date** with the *underlying data*
  - is known as *materialized view* ***maintenance***
- Materialized views **can** be maintained by **recomputation** on *every update*
- A **better** *option* is:
    - to use **incremental view maintenance**
  - **Changes** to *database relations* are **used**
    - to **compute** *changes to materialized view*,
    - which is then **updated**
- View maintenance **can** be done by:
  - Manually **defining triggers** on insert, delete, and update of each relation in the view definition

- Manually **written code** to update the view whenever database relations are updated
- **OR**: Supported **directly by the database**

***Incremental* View Maintenance**

- The **changes** (inserts and deletes) to a relation or expressions are
  - *referred to* as its **differential**
  - ***Set of tuples*** inserted to and deleted from **r** are
    - denoted $\mathbf{i_r}$ and $\mathbf{d_r}$
- To *simplify* our description, we *only consider* inserts and deletes
  - We replace ***updates*** to a tuple by
    - *deletion* of the tuple followed by
    - *insertion* of the update tuple
- We describe how to **compute** the change:
  - to the **result of** *each* relational *operation*,
    - **given** changes to its ***inputs***
- We then **outline** *how to handle* relational algebra *expressions*

TRANSACTIONS

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability

Transaction *Concept*

- A **transaction** is a *unit* of program execution that
  - ***accesses*** and possibly ***updates*** various data items.
- A transaction **must** see a ***consistent*** *database*.
  - During transaction **execution**:
    - the database may be temporarily *inconsistent*.
  - When the transaction **completes** successfully (is committed),
    - the database **must** be consistent.
  - After a transaction commits,
    - the **changes** it has made to the database **persist**,
      - even if there are *system failures*.
- **Multiple** transactions **can** execute in parallel.
  - **Two main issues** to deal with:
    - Failures of various kinds, such as :
      - hardware *failures* and system *crashes*

- Concurrent execution of *multiple* transactions

ACID Properties

- A **transaction** is a *unit* of program execution that:
    - ***accesses*** and possibly ***updates*** various data items.
- To **preserve** the **integrity** of data, the database system must **ensure**:
    - **Atomicity.** Either all operations of the transaction are:
        - properly reflected in the database or none are.
    - **Consistency.** Execution of a transaction in isolation:
        - **preserves** the **consistency** of the database.
    - **Isolation.** Although multiple transactions may execute concurrently,
        - **each** *transaction* must be **unaware** of other concurrent *transactions*.
        - Intermediate transaction results ***must be***:
            - ***hidden*** from other concurrently executed transactions.
        - That is, for every pair of transactions T_i and T_j, it appears to T_i that **either** T_j, finished execution before T_i started, **or** T_j started execution after T_i finished.
    - **Durability.** After a transaction ***completes*** successfully,
        - the **changes** it has made to the database **persist**,
            - even if there are system *failures*.

Example of *Fund Transfer*

- Transaction to transfer $50 from account **A** to account **B**:

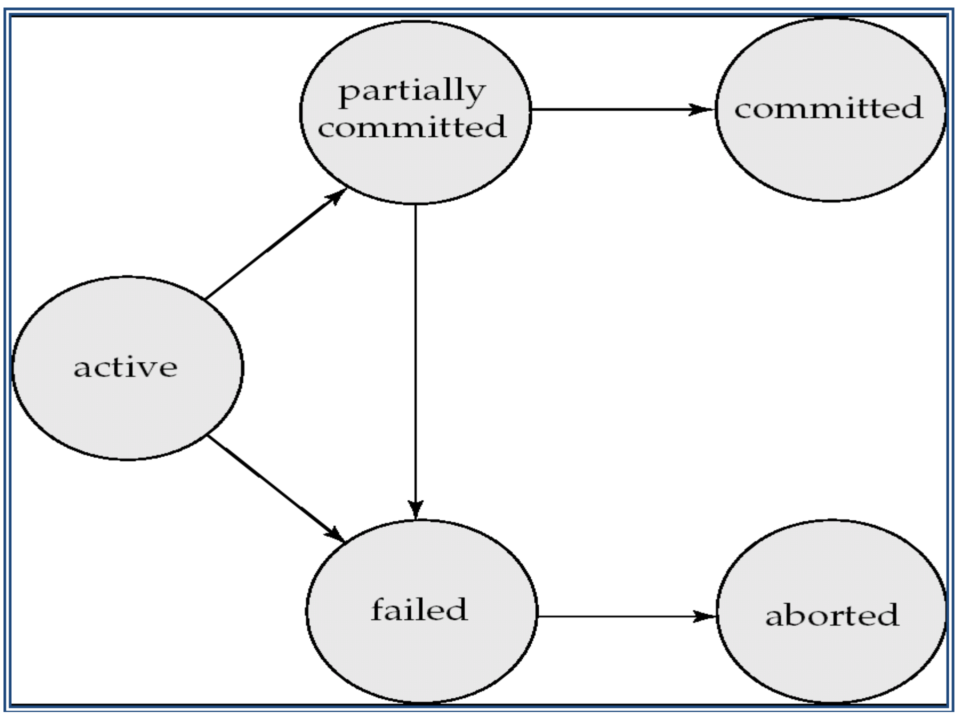
1. **read**(A)
2. $A := A – 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

- **Atomicity requirement** :
    - if the transaction **fails** after step 3 and before step 6,
        - the **system** should **ensure** that :
            - its **updates** are *not reflected* in the database,
            - else an *inconsistency* will result.
- **Consistency requirement** :
    - the **sum** of **A** and **B** is:
        - **unchanged** by the execution of the transaction.
- **Isolation requirement** —
    - if between steps 3 and 6,
        - another transaction is allowed to access the partially updated database,
            - it will see an inconsistent database
            - (the sum $A + B$ will be less than it should be).
    - Isolation can be **ensured** trivially by:
        - running transactions **serially**,
            - that is **one** after the **other**.

- *However*, executing multiple *transactions* **concurrently**
  - has significant **benefits**, as we will see later.
- **Durability requirement** :
  - once the user has been notified that the transaction has **completed** :
    - (i.e., the transfer of the $50 has taken place),
    - the **updates** to the database by the transaction **must persist**
      - despite *failures*.

Transaction *State*

- **Active** – the initial state;
  - the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that ***normal execution*** can no longer proceed.
- **Aborted** – after the transaction has been rolled back and
  - the database restored to its state prior to the start of the ***transaction***.
  - **Two options** after it has been aborted:
    - restart the transaction; can be done
      - only **if** no internal logical error
    - **kill** the transaction
- **Committed** – after **successful** completion.

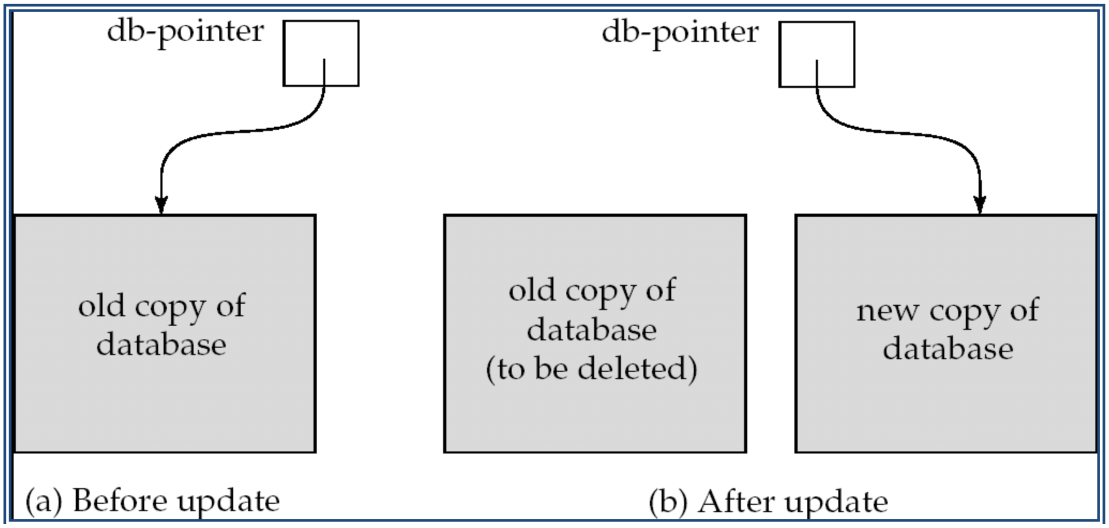


Implementation **of Atomicity and Durability**

- The recovery-management component of a database system
  - implements the support for atomicity and durability.
- The *shadow-database* scheme:

    - assume that only one *transaction* is active at a time.
    - a pointer called db_pointer always points to the current consistent copy of the database.
    - all updates are made on a *shadow copy* of the database, and db_pointer is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
    - in case transaction fails, old consistent copy pointed to by db_pointer can be used, and the shadow copy can be deleted.

- Assumes **disks** do not fail
- Useful for **text editors**, but
    - extremely **inefficient** for *large databases* (why?)
    - Does **not** handle **concurrent** *transactions*



- Assumes **disks** do not fail
- Useful for **text editors**, but
    - extremely **inefficient** for *large databases* (why?)
    - Does **not** handle **concurrent** *transactions*

***Concurrent* Executions**

- Multiple transactions are allowed to run concurrently in the system:
    - **increased processor and disk utilization**,
        - leading to **better** *transaction **throughput**:*
            - **one** *transaction* can be using the CPU while
            - **another** is reading from or writing to the disk
    - **reduced average response time** for transactions:
        - ***short*** *transactions* need not wait behind ***long*** **ones**.
- **Concurrency control schemes** :
    - mechanisms to achieve isolation; that is,
        - to **control** the **interaction** among the concurrent *transactions*
    - in order to **prevent** them from destroying the **consistency** of the database
        - Will study in Chapter 16, after studying

– notion of correctness of *concurrent executions*.

Schedules

- **Schedule** – a sequences of instructions that **specify** the chronological order
    - in which ***instructions*** of concurrent transactions are ***executed***
  - a schedule for a ***set of transactions*** must consist of
    - *all instructions* of those *transactions*
  - must **preserve** the **order** in which
    - the instructions **appear** in each individual transaction.
- A ***transaction*** that successfully ***completes*** its execution
  - will have a **commit** *instructions* as the last statement
    - (will be omitted if it is obvious)
- A ***transaction*** that **fails** to successfully complete its execution
  - will have an **abort** *instructions* as the last statement
    - (will be omitted if it is obvious)

Schedule 1

- Let :
    - **T_1** transfer **$50** from A to B, and
    - **T_2** transfer **10%** of the balance from A to B.
- A **serial** schedule in which **T_1** is followed by **T_2**:

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

T_1	T_2
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B)
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B)	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously**:**
  - The following schedule is **not** a **serial** schedule,
  - but it is ***equivalent*** to Schedule 1.

T_1	T_2
read(A) $A := A - 50$ write(A)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B)	
	read(B) $B := B + temp$ write(B)

- In Schedules **1**, **2** and **3**:
  - the **sum(A + B)** is ***preserved***.

Schedule 4

T_1	T_2
read(A) $A := A - 50$	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B)
write(A) read(B) $B := B + 50$ write(B)	
	$B := B + temp$ write(B)

Serializability

- **Basic Assumption**:
    - Each **transaction preserves** database **consistency**.
- Thus:
    - **Serial** execution of a **set of transactions preserves** database **consistency**.
- A (*possibly concurrent*) **schedule** is ***serializable*** :
    - **if** it is **equivalent** to a ***serial*** *schedule*.
- Different forms of ***schedule equivalence*** give rise to the notions of:

1. **conflict** serializability
2. **view** serializability

- We ignore operations other than **read** and **write** instructions, and we assume that:
    - transactions may perform:
        - arbitrary computations on data in local buffers,
        - in between **reads** and **writes**.
    - Our *simplified schedules* consist of :
        - only **read** and **write** instructions.

***Conflicting* Instructions**

- Instructions l_i and l_j of transactions T_i and T_j respectively,
    - **conflict** if and only if there exists
        - some item Q accessed by **both** l_i and l_j, and
        - at least one of these instructions **wrote** Q.

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

- Intuitively, a **conflict** between l_i and l_j **forces**
    - a (logical) **temporal *order*** between them.

- If l_i and l_j are consecutive in a schedule and they do not conflict,
    - their **results** would remain the **same**
        - even if they had been interchanged in the *schedule*.

Conflict *Serializability*

- **If** a *schedule* ***S*** can be **transformed** into a *schedule* ***S´*** by
    - a *series of* ***swaps*** of **non-conflicting** instructions,
        - we say that ***S*** and ***S´*** are **conflict equivalent**.
- We say that a *schedule* ***S*** is ***conflict serializable***
    - **if** it is *conflict equivalent* to a **serial** schedule
- Schedule **3** can be transformed into *Schedule* **6**,
    - a *serial schedule* where T_2 follows T_1,
        - by *series of* ***swaps*** of ***non-conflicting*** instructions.
    - Therefore:
        - Schedule **3** is ***conflict serializable***.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

- Example of a **schedule** that is **not** *conflict serializable*:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- We are **unable** to **swap** *instructions* in the above schedule to obtain:
    - either the serial schedule < T_3, T_4 >,
    - or the serial schedule < T_4, T_3 >.

View *Serializability*

- Let ***S*** and ***S´*** be two schedules with the **same** set of transactions.
  - ***S*** and ***S´*** are **view equivalent**, **if** the following **three** conditions are met:

1. For each data item ***Q***, **if** transaction T_i **reads** the initial value of ***Q*** in schedule ***S***,

- then transaction T_i **must**, in schedule ***S´***, also **read** the initial value of ***Q***.

2. For each data item ***Q*** if transaction T_i executes **read(*Q*)** in schedule ***S***, and that **value** was produced by transaction T_j (if any),

- then transaction T_i must in schedule ***S´*** also **read** the value of ***Q*** that was produced by transaction T_j .

3. For each data item ***Q***, the transaction (if any) that **performs** the final **write**(***Q***) operation in schedule ***S***

- **must perform** the final **write(*Q*)** operation in schedule *S´*.

As can be seen, **view** equivalence is also based purely on **reads** and **writes** alone.

- A schedule ***S*** is **view serializable**,
  - **if** it is view equivalent to a ***serial*** *schedule*.
- ***Every*** conflict serializable schedule is **also** view serializable.
- Below is a schedule which is view-serializable but ***not*** conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- What ***serial*** *schedule* is above equivalent to?
- Every view serializable schedule that is **not** conflict serializable**:**
  - has ***blind writes*.**

Other Notions of *Serializability*

- The schedule below produces **same** outcome as the **serial** schedule < T_1, T_5 >,
  - yet**!**, is **not** *conflict equivalent* or *view equivalent* to it**!**

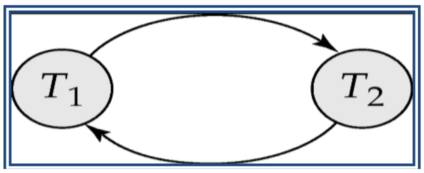
T_1	T_5
read(A) $A := A - 50$ write(A)	
	read(B) $B := B - 10$ write(B)
read(B) $B := B + 50$ write(B)	
	read(A) $A := A + 10$ write(A)

- ***Determining*** *such equivalence* requires:

    - ***analysis** of **operations*** other than **read** and **write**.

Testing for *Serializability*

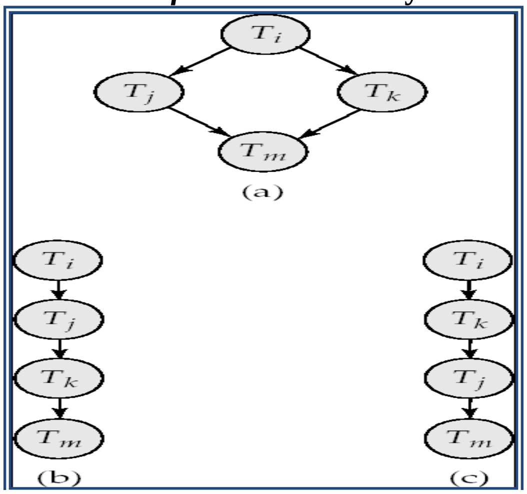
- Consider some schedule of a set of transactions $\boldsymbol{T_1, T_2, ..., T_n}$
- **Precedence graph** :
    - a ***direct graph*** where the vertices ***are*** the *transactions* (names).
- We draw an arc from $\boldsymbol{T_i}$ to $\boldsymbol{T_j}$
    - if the two transaction conflict, and
    - $\boldsymbol{T_i}$ accessed the data item on which the conflict arose **earlier**.
- We may label the arc by the **item** that was **accessed**.
- ***Ex*. 1:**



Example Schedule (Schedule A) + *Precedence Graph*

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

Test for _Conflict_ Serializability



- A schedule is *conflict serializable*
    - **if** and only **if** its *precedence graph is* ***acyclic***.
- **Cycle-detection** algorithms exist which:
    - take **order n^2** time,
        - where ***n*** is the *number of vertices* in the graph.
    - Better algorithms take **order $n + e$**
        - where ***e*** is the *number of edges*.
- **If** precedence graph is **acyclic**, the *serializability order* can be obtained by a ***topological sorting*** of the graph.
    - This is a linear order *consistent* with:
        - the partial order of the graph.
    - For example, a *serializability order* for Schedule A would be:

      $$T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$$

        - Are there others?

Test for _View_ Serializability

- The precedence graph test for conflict serializability
    - **cannot** be **used** *directly* to test for **view** serializability.
    - Extension to test for *view serializability* has:
        - **cost *exponential*** in the size of the precedence graph.
- The problem of **checking** if a schedule is **view** serializable:
    - falls in the class of:
        - ***NP*-complete** problems.
    - Thus existence of an efficient algorithm is:
        - *extremely* **unlikely**.
- However practical algorithms that:
    - just check some **sufficient conditions** for view serializability
        - **can** still **be used**.

Recoverable Schedules

Need to address the effect of *transaction **failures*** on concurrently *running transactions*:

- **Recoverable schedule**:
  - if a transaction T_j **reads** a data item *previously* **written** by a transaction T_i ,
  - then the **commit** operation of T_i appears **before** the **commit** operation of T_j.
- The following schedule (Schedule 11) is **not** recoverable
  - if T_9 **commits** immediately after the read

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- If T_8 should **abort**, T_9 would:
  - **have read** (and possibly shown to the user) *an **inconsistent** database state*.
- Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback**:
  - a **single** *transaction failure* **leads** to a series of transaction rollbacks.
  - Consider the following schedule where:
    - **none** of the transactions has yet **committed**

(so the schedule is **recoverable**)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

  - If T_{10} *fails*, T_{11} and T_{12} must also be *rolled back*.
- Can **lead** to the *undoing* of a *significant amount of work*

Cascadeless Schedules

- **Cascadeless schedules**:
  - cascading rollbacks cannot occur;
    - for each pair of transactions T_i and T_j such that
      - T_j **reads** a data item previously **written** by T_i,
      - the **commit** operation of T_i appears:
        - » before the **read** operation of T_j.
- Every **cascadeless** schedule:
  - is also **recoverable**
- It is ***desirable*** to **restrict** the schedules to:

- those that are ***cascadeless***

Concurrency **Control**

- A database must **provide** a mechanism that
  - will ensure that **all** possible schedules are:
    - either conflict or view serializable, and
    - are recoverable and
      - *preferably* cascadeless
- A policy in which only one transaction can execute at a time :
    - generates serial schedules,
    - but provides a *poor degree of concurrency*
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability:
  - *after* it has executed
    - is a little too late!
- **Goal**:
  - to develop concurrency control protocols that:
    - will ***assure*** *serializability*.

Concurrency Control vs Serializability Tests

- Concurrency control protocols:
  - allow concurrent schedules,
    - but **ensure** that the schedules are conflict / view serializable, and
    - are recoverable and cascadeless.
- Concurrency control protocols generally
  - do not examine the precedence graph as it is being created
  - Instead a protocol **imposes** a **discipline** that
    - avoids nonseralizable schedules.
  - We study such protocols in Chapter 16.
- **Different** concurrency control protocols:
    - provide **different** tradeoffs between
      - the *amount of concurrency* they allow and
      - the *amount of overhead* that they incur.
- Tests for serializability **help** us understand

why a concurrency control protocol is **correct**.

Weak Levels **of Consistency**

- Some applications are willing to live with weak levels of consistency,
  - **allowing** schedules that are not serializable
  - *E.g*. a *read-only transaction* that wants to
    - get an approximate total balance of all accounts
  - E.g. database statistics computed
    - for query optimization can be approximate (why?)
  - Such transactions :
    - need not be serializable with respect to other transactions
- Tradeoff :
    - accuracy for

- performance

***Levels of Consistency* in SQL**

- **Serializable** : default
- **Repeatable read** :
    - **only** committed records to be read,
    - **repeated reads** of same record must return same value.
  - However, a transaction **may not** be serializable –
    - it may **find** some records **inserted** by a transaction
    - but **not find others!**
- **Read committed** :
    - **only** committed records can be read,
  - but **successive reads** of record
    - may return different (but committed) values.
- **Read uncommitted** :
    - even uncommitted records may be read.
- Lower degrees of consistency useful for:
    - gathering *approximate information* about the database

Transaction Definition in SQL

- Data manipulation language must include a construct for:
    - specifying the **set of actions** that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- Levels of consistency specified by **SQL-92**:
  - **Serializable** — default
  - **Repeatable read**
  - **Read committed**
  - **Read uncommitted**

CONCURRENCY CONTROL

- **Lock-Based** Protocols
- **Timestamp-Based** Protocols
- **Validation-Based** Protocols
- Multiple **Granularity**
- **Multiversion** Schemes
- **Deadlock** Handling
- Insert and Delete **Operations**
- Concurrency in **Index** Structures

***Lock-Based* Protocols**

- A **lock** is a ***mechanism*** to *control concurrent access* to a data item
- Data items can be locked in *two modes* :

1. ***exclusive*** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
2. ***shared*** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

- Lock requests are made to *concurrency-control* ***manager***.
  - Transaction **can proceed** only after *request* is granted.

- ***Lock-compatibility*** matrix

	S	X
S	true	false
X	false	false

- A ***transaction*** may be granted a lock on an item
  - **if** the *requested lock* is ***compatible*** with
    - locks already held on the item by ***other transactions***
- Any number of transactions can hold **shared** locks on an item,
  - **but** if any transaction holds an **exclusive** on the item
    - no other transaction may hold any lock on the item.
- If a lock cannot be granted,
  - the requesting transaction is made to **wait** till:
    - **all** incompatible locks held by other transactions have been released.
      - The lock is then granted.
- Example of a transaction performing locking:

  T_2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A+B*)

- Locking as above is ***not sufficient*** to guarantee serializability:
  - if *A* and *B* get updated in-between the read of *A* and *B*,
    - the displayed sum would be wrong.

- A **locking protocol** is a set of rules *followed by*
  - **all transactions** while requesting and releasing locks.
- Locking protocols **restrict** the set of possible **schedules**.

***Pitfalls* of Lock-Based Protocols**

- Consider the **partial** schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress:
  - executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while
  - executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is *called* a **deadlock**.
  - To **handle** a deadlock one of T_3 or T_4 **must** be rolled back
    - and its locks released.
- The potential for deadlock exists in:
    - **most** locking **protocols**.
  - *Deadlocks* are a *necessary evil*.
- **Starvation** is also ***possible***
    - **if** *concurrency control manager* is ***badly designed***.

For example:

  - A transaction may be waiting for an **X-lock** on an item,
    - while a ***sequence*** of other ***transactions***
      - request and are granted an **S-lock** on the same item.
  - The **same transaction** is
    - repeatedly rolled back due to deadlocks.
- *Concurrency control manager can be designed* to:
    - **prevent** starvation.

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing** Phase
  - transaction may obtain locks
  - transaction *may not release* locks
- Phase 2: **Shrinking** Phase
  - transaction may release locks
  - transaction *may not obtain* locks
- The protocol **assures** serializability.
  - It can be proved that the transactions can be
    - serialized *in the order of* their **lock points**
    - i.e. the point where a transaction acquired its final lock.
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking.
  - To avoid this, follow a modified protocol :
    - called **strict two-phase locking**.
      - Here a transaction must hold **all** its ***exclusive locks*** :
      - till it commits/aborts.
- **Rigorous two-phase locking** is even stricter:
  - here ***all* locks** are **held:**
    - till commit/abort.
  - In this protocol transactions can be serialized
    - *in the order* in which they commit.
- There can be **conflict** serializable schedules that:
  - cannot be obtained **if** *two-phase locking* is used**!**
- However, in the ***absence*** of *extra information* (e.g., ordering of access to data),
  - *two-phase locking* is needed for:
    - **conflict** serializability in the following sense:
      - Given a transaction T_i that does not follow two-phase locking,
      - we can find a transaction T_j that uses two-phase locking,
      - and a schedule for T_i and T_j that
        - n **is not** conflict serializable.

Lock *Conversions*

- ***Two-phase locking*** with *lock **conversions***:

– First Phase:

    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures ***serializability***.
- But still **relies on** the ***programmer***
  - to insert the *various* locking instructions.

Automatic **Acquisition of Locks**

- A transaction T_i issues the standard read/write instruction,
  - without *explicit locking calls*.
- The operation **read**(D) is processed as:

if T_i has a lock on D
  then
    read(D)
  else begin
    if necessary wait until no other
      transaction has a **lock-X** on D
    grant T_i a **lock-S** on D;
    read(D)
  end

- **write**(D) is processed as:

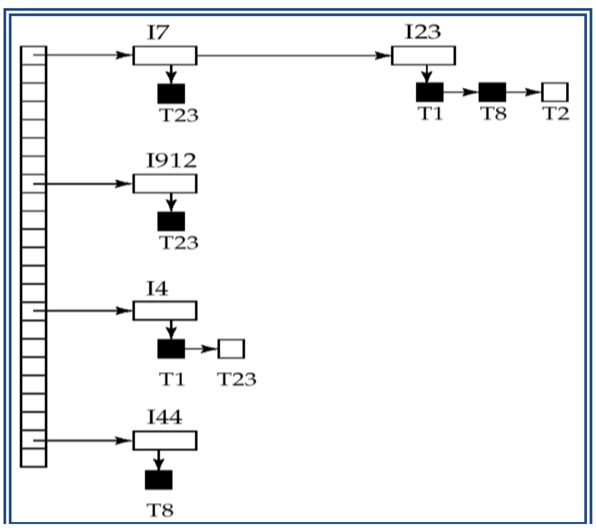
if T_i has a **lock-X** on D
  then
    write(D)
  else begin
    if necessary wait until no other trans. has any lock on D,
    if T_i has a **lock-S** on D
      then
        upgrade lock on D to **lock-X**
      else
        grant T_i a **lock-X** on D
      write(D)
  end;

- All locks are released after commit or abort

***Implementation* of Locking**

- A **lock manager** can be implemented as:
  - a separate process to which:
    - transactions send lock and unlock requests
- The lock manager **replies** to a lock request by
  - sending a lock grant messages
  - or a message asking the transaction to roll back, in case of a deadlock.
- The requesting transaction waits:
    - until its request is answered
- The lock manager maintains a data-structure called a **lock table** to
    - record granted locks and pending requests
- The lock table is usually *implemented* as:
  - an in-memory ***hash table*** indexed on:
    - the **name** of the data item being locked

Lock *Table*



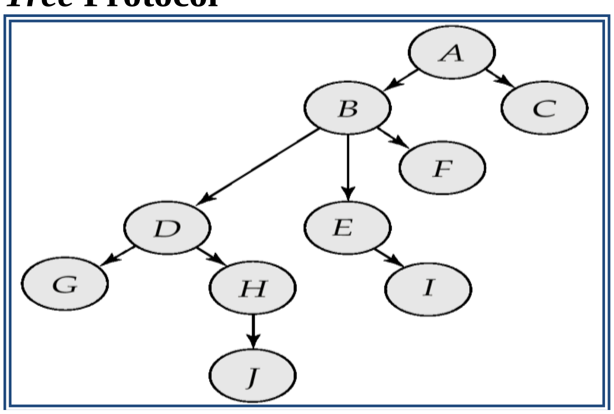
- ***Black rectangles*** indicate granted locks,
  - ***white ones*** indicate waiting requests
- Lock table also records:
    - the ***type of lock*** granted or requested
- New request is added to the end of:
  - the **queue** of requests for the data item, and
  - granted **if** it is **compatible** with all earlier locks

- Unlock requests result in
  - the request being deleted, and
  - later requests are checked
    - to see **if** they can now be granted
- If transaction ***aborts***,
- **all** waiting or granted **requests** of the transaction are **deleted**
- lock manager may keep a *list of locks* held by each transaction,
- to implement this *efficiently*

Graph-Based Protocols

- **Graph-based** protocols are:
  - an ***alternative*** to *two-phase locking*
- **Impose** a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, ..., d_h\}$ of ***all*** data ***items***.
  - If $d_i \rightarrow d_j$ then:
    - any transaction accessing both d_i and d_j
    - must access d_i *before* accessing d_j.
  - Implies that the set **D** may now be viewed as
    - a directed acyclic graph,
    - called a ***database graph***.
- The ***tree-protocol*** is:
    - a simple kind of graph protocol.

Tree Protocol



- **Only exclusive** locks are allowed.
- The **first** lock by T_i may be on any data item.
  - Subsequently, a ***data Q*** can be locked by T_i
    - ***only if*** the **parent of** Q is currently locked by T_i.
- ***Data items*** may be unlocked at **any time**.
- The tree protocol ensures:

        - **conflict** serializability as well as freedom from ***deadlock***.
- Unlocking may occur earlier in the tree-locking protocol
        - than in the *two-phase locking* protocol.
    - shorter waiting times, and increase in *concurrency*
    - protocol is deadlock-free, no rollbacks are required
- Drawbacks
    - Protocol does not guarantee *recoverability* or *cascade freedom*
        - Need to introduce ***commit dependencies***
            - to ensure recoverability
    - Transactions may have to lock data **items** that they do not access.
        - increased locking overhead, and additional waiting time
        - potential decrease in concurrency
- **Schedules** not possible under *two-phase locking* are:
    - possible under tree protocol, and vice versa.

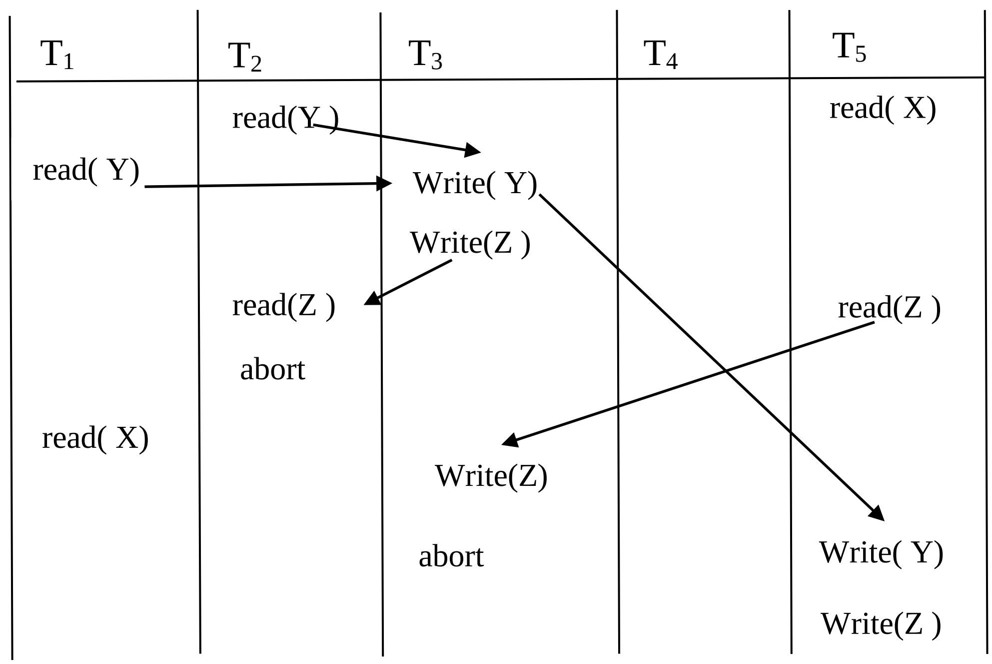
Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system.
    - If an old transaction T_i has time-stamp TS(T_i),
        - a new transaction T_j is assigned time-stamp TS(T_j)
            - such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that
    - the time-stamps determine the serializability order.
- In order to assure such behavior,
    - the protocol maintains for each data Q **two timestamp** values:
        - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
        - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.
- The **timestamp ordering** protocol ensures that:
    - any conflicting **read** and **write** operations are:
        - executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q):
    - If TS(T_i) < **W**-timestamp(Q), then T_i needs to ***read*** a value of Q that was already overwritten.
        - Hence, the **read** operation is **rejected**, and T_i is rolled back.
    - If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is **executed**, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).
- Suppose that transaction T_i issues **write**(Q):

- If TS(T_i) < **R**-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the **write** operation is **rejected**, and T_i is rolled back.
- If TS(T_i) < **W**-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
    - Hence, this **write** operation is **rejected**, and T_i is **rolled back**.
- Otherwise, the **write** operation is **executed**, and W-timestamp(Q) is set to TS(T_i).

Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5



Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol ***guarantees*** *serializability*
    - since **all** the arcs in the precedence graph are of the form:

Thus, there will be **no cycles** in the precedence graph

- Timestamp protocol **ensures** *freedom from deadlock* as
  - **no** transaction ever **waits**.
- But the schedule may **not** be *cascade-free*, and
  - may **not** even be *recoverable*.

Recoverability **and** ***Cascade*** **Freedom**

- Problem with timestamp-ordering protocol:
  - Suppose $\boldsymbol{T_i}$ aborts, but $\boldsymbol{T_j}$ *has read a data* item ***written by*** $\boldsymbol{T_i}$
  - Then $\boldsymbol{T_j}$ must abort:
    - ***if*** $\boldsymbol{T_j}$ had been *allowed to commit earlier*,
      - the schedule is not recoverable!
    - ***otherwise***, *any* $\boldsymbol{T_k}$ that *has read a data* item ***written by*** $\boldsymbol{T_j}$ must abort too!
      - This can lead to cascading rollbacks!
- Solution 1:
  - A transaction is structured such that:
    - **all** its **writes** are performed **at** the **end** of its processing
  - **All** writes of a transaction form an **atomic** action:
    - no transaction may **execute** while a transaction is being written
  - A **transaction** that aborts is restarted with a **new timestamp**
- Solution 2: Limited form of locking: wait for data to be **committed** before reading it
- Solution 3: Use **commit dependencies** to ensure recoverability

Thomas' Write Rule

- *Modified version* of the timestamp-ordering protocol in which:
  - obsolete **write** operations *may be* ***ignored*** under certain circumstances.
- When T_i attempts to write data item Q,
  - if TS(T_i) < W-timestamp(Q),
    - then T_i is *attempting to write* an *obsolete value* of {Q}.
  - Rather than *rolling back* T_i :
    - as the *timestamp ordering* protocol would have done,
    - this {**write**} operation can be ignored.
- ***Otherwise,*** this protocol is the same as the timestamp ordering protocol.

- Thomas' Write Rule **allows** *greater* potential *concurrency*:
  - **Allows** some *view-serializable* schedules that:
    - ***are not*** *conflict-serializable*.

Validation-Based **Protocol**

- Execution of transaction T_i is done in ***three phases***:

1. **Read and execution** phase: Transaction T_i writes:
   - only to *temporary local variables*
2. **Validation** phase: Transaction T_i performs a "**validation** test"‘:
   - to determine if local variables can be written *without violating* ***serializability***.
3. **Write** phase: If T_i is **validated**, the updates are applied to the database
   - otherwise, T_i is **rolled back**!

- The **three phases** of concurrently executing transactions can be **interleaved**,
  - but each transaction must go through the three phases in that order.
  - Assume for simplicity that:
    - the validation and write phase ***occur together***, *atomically* and *serially*
    - I.e., only one transaction **executes** validation/write **at a time**.
- Also called as **optimistic concurrency control** since:
  - transaction ***executes fully*** in the hope that all will go well during validation
- Each transaction T_i has **3** timestamps:
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is **determined** by:
    - timestamp given at validation time,
    - to **increase** concurrency.
  - Thus TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and
    - gives **greater** degree of concurrency
    - **if** probability of conflicts is **low**.

because:

- the serializability order is **not** pre-decided, and
- relatively **few** transactions will have to be rolled back.

***Validation Test* for Transaction T_j**

- **If** for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j) or
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and**
    - the set of data items written by T_i does not intersect with
    - the set of data items read by T_j.

  then validation succeeds and T_j can be **committed**.
- ***Otherwise***, validation **fails** and T_j is **aborted**.

***Justification*:**

- Either the first condition is satisfied, and
  - there is **no** overlapped execution,
- Or, the second condition is satisfied and
  - the writes of T_j do **not** affect reads of T_i since
    - they *occur after* T_i has finished its reads.
  - the writes of T_i do **not** affect reads of T_j since
    - T_j does *not read* any item written by T_i.

Schedule Produced by *Validation*

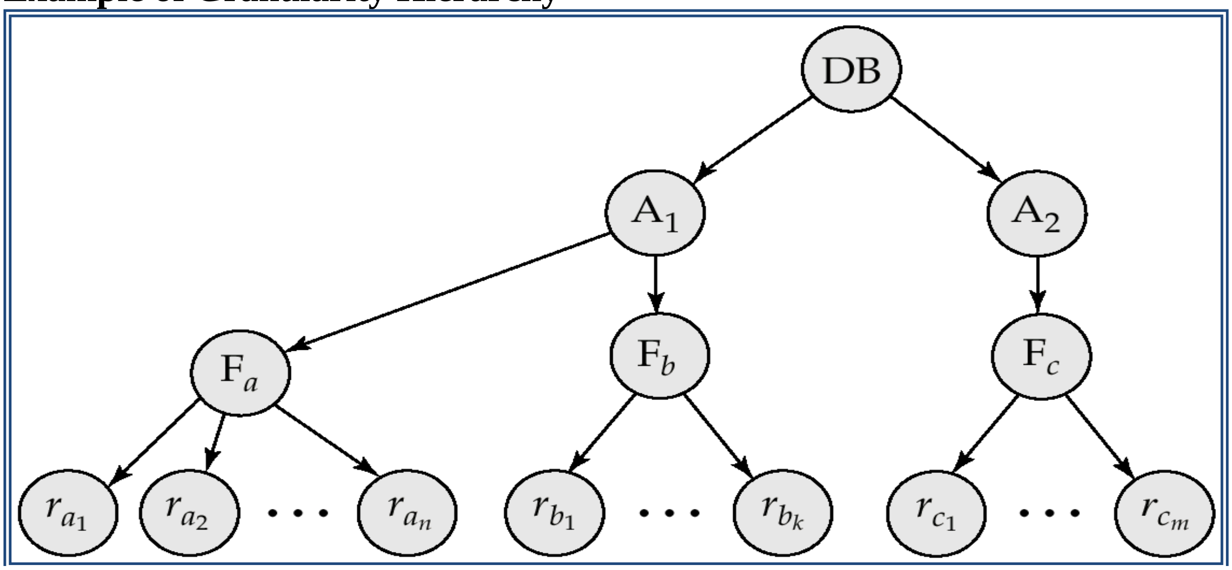
- Example of schedule produced using validation

T_{14}	T_{15}
read(B)	
	read(B)
	B:= B-50
	read(A)
	A:= A+50
read(A)	
(*validate*)	
display (A+B)	
	(*validate*)
	write (B)
	write (A)

Multiple *Granularity*

- Allow **data items** to be of **various sizes** and
  - **define** a hierarchy of data granularities, where :
    - the small granularities are **nested** within larger ones
- Can be represented graphically as a **tree**
  - but **don't confuse** with *tree-locking* protocol
- When a transaction **locks** a node in the tree *explicitly*,
  - it ***implicitly*** **locks** all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine** granularity (lower in tree):
    - high concurrency, high locking overhead
  - **coarse** granularity (higher in tree):
    - low locking overhead, low concurrency

Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes

- In addition to **S** and **X** ***lock*** *modes*, there are three additional lock modes with multiple granularity:

- ***intention-shared*** (**IS**): indicates explicit locking at a lower level of the tree but only with shared locks.
- ***intention-exclusive*** (**IX**): indicates explicit locking at a lower level with exclusive or shared locks
- ***shared and intention-exclusive*** (**SIX**):
  - the subtree rooted by that node is locked explicitly in shared mode and
  - explicit locking is being done at a lower level with exclusive-mode locks.
- **intention locks allow** a higher level node to **be locked** in **S** or **X** mode
  - without having to check ***all descendent*** nodes.

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

	I	I	S	S IX	X
I	✓	✓	✓	✓	×
I	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

Multiple Granularity Locking *Scheme*

- Transaction $\boldsymbol{T_i}$ *can lock* a node ***Q***, using the *following* ***rules***:
  1. The lock compatibility matrix ***must be*** observed.
  2. The root of the tree ***must be*** locked ***first***, and may be locked in *any mode*.
  3. A node *Q* can be ***locked*** by T_i in **S** or **IS** *mode* **only if:**

- the parent of Q is *currently locked* by T_i in either **IX** or **IS** *mode*.

4. A node Q can be ***locked*** by T_i in **X**, **SIX**, or **IX** *mode*:
   - only if the parent of Q is *currently locked* by T_i in either **IX** or **SIX** *mode*.
5. T_i *can* ***lock*** a node **only if**:
   - it has **not** *previously* ***unlocked*** *any node* (i.e., T_i is **two-phase**).
6. T_i **can unlock** a node Q **only if**:
   - **none** *of the children* of Q are *currently locked* by T_i.

- Observe that locks are ***acquired*** in root-to-leaf order,
  - whereas they are ***released*** in leaf-to-root order.

Multiversion **Schemes**

- **Multiversion** schemes **keep** old versions of data item to increase ***concurrency***.
  - Multiversion *Timestamp* Ordering
  - Multiversion *Two-Phase* Locking
- Each successful **write** *results in*:
  - the ***creation*** of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(Q) operation is issued,
  - ***select*** an appropriate version of Q *based on* the timestamp of the transaction, and return the value of the selected version.
- **read**s ***never*** *have to* ***wait*** as an appropriate version is returned immediately.

Multiversion *Timestamp* Ordering

- Each data item Q has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k
- when a transaction T_i creates a new version Q_k of Q,
  - Q_k's W-timestamp and R-timestamp are initialized to **TS(T_i)**.
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and:
  - TS(T_j) > R-timestamp(Q_k).
- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation.

- Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).
- If transaction T_i issues a **read**(Q), then the value returned is the content of version Q_k.
- If transaction T_i issues a **write**(Q)
  - if TS(T_i) < R-timestamp(Q_k), then transaction T_i is **rolled back**.
  - if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
  - else a new version of Q is created.

- Observe that
  - Reads always succeed
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write,
    - has already read a version created by a transaction older than T_i.
- Protocol guarantees **serializability**

Multiversion *Two-Phase* Locking

- Differentiates between read-only *transactions* and update *transactions*
- ***Update*** *transactions* **acquire** read and write **locks**, and:
  - **hold all locks** up to the end of the transaction.
    - That is, update transactions follow ***rigorous*** *two-phase locking*.
  - Each successful **write** results in:
    - the creation of a new version of the data item written.
  - each version of a data item has a **single** timestamp:
    - whose value is obtained from a counter **ts-counter** that is:
      - incremented during **commit** processing.
- ***Read-only*** *transactions* are assigned a timestamp by :
  - reading the current value of **ts-counter** before they **start** execution;
  - they **follow** the *multiversion timestamp-ordering* protocol:
    - for performing reads.
- When an update *transaction* wants to read a data item:
  - it obtains a shared lock on it, and reads the *latest version*.
- When it wants to write an item:
  - it obtains **X-lock** on;
  - it then creates a new version of the item and
    - **sets** this version's timestamp to ∞.
- When update transaction T_i completes, commit processing occurs:
  - T_i **sets** timestamp on the versions it has created to **ts-counter** + 1

- T_i **increments ts-counter** by 1
- Read-only transactions that **start** after $\boldsymbol{T_i}$ increments **ts-counter**:
  - **will see** the values **updated** by $\boldsymbol{T_i}$.
- Read-only transactions that **start** before $\boldsymbol{T_i}$ increments the **ts-counter:**
  - will see the value before the **updates** by $\boldsymbol{T_i}$.
- Only ***serializable*** *schedules* are produced.

Deadlock Handling

- Consider the following two transactions:

  T_1: write (X) write(Y) T_2: write(Y) write(X)

- Schedule with **deadlock**

T_1	T_2
X-lock on X write (X)	
	X-lock on Y write (X) wait for **X-lock** on X
wait for **X-lock** on Y	

Deadlock Handling

- System is **deadlocked** if there is a set of transactions such that:
  - every transaction in the set **is waiting** for another transaction in the set.
- ***Deadlock prevention*** protocols **ensure** that:
  - the system will ***never*** enter into a deadlock state.
- Some **prevention** strategies :
  - **Require** that each transaction **locks all** its data items:
    - before it begins **execution** (**predeclaration**).
  - **Impose** partial **ordering** of **all data items** and require that:
    - a transaction can lock data items:
      - only in the order ***specified*** by the partial **order**
      - (graph-based protocol).

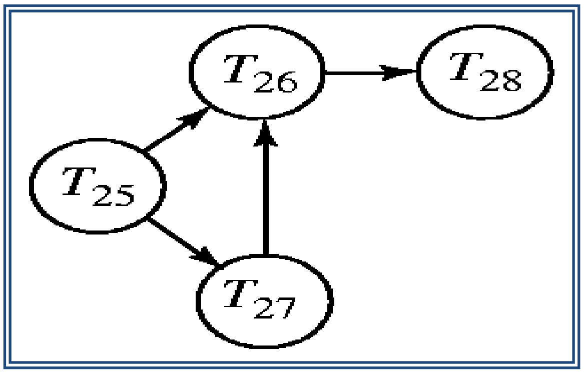
More Deadlock *Prevention* Strategies

- Following schemes use transaction timestamps:
  - for the sake of deadlock **prevention** alone.
- **wait-die** scheme — *non-preemptive*
  - **older** transaction may **wait** for younger one to release data item.
  - Younger transactions **never** wait for **older** ones;
    - they are **rolled back** instead.
  - a transaction may die several times before **acquiring** needed data item
- **wound-wait** scheme — *preemptive*
  - **older** transaction ***wounds*** (**forces** rollback) of younger transaction:
    - instead of waiting for it.
  - Younger transactions may **wait** for **older** ones.
  - may be **fewer** *rollbacks* than ***wait-die*** scheme.
- Both in ***wait-die*** and in ***wound-wait*** schemes,
  - a rolled back transactions is **restarted** with its **original** timestamp.
  - **Older** transactions thus have **precedence** over newer ones,
  - and starvation is hence **avoided**.
- **Timeout-Based** Schemes :
  - a transaction waits for a **lock**:
    - only for a specified amount of **time**.
    - After that, the wait **times out** and:
      - n the transaction is rolled back.
  - thus deadlocks are **not** possible
  - **simple** to implement;
  - but starvation is **possible**.
  - Also difficult to determine ***good value*** of the **timeout** interval.

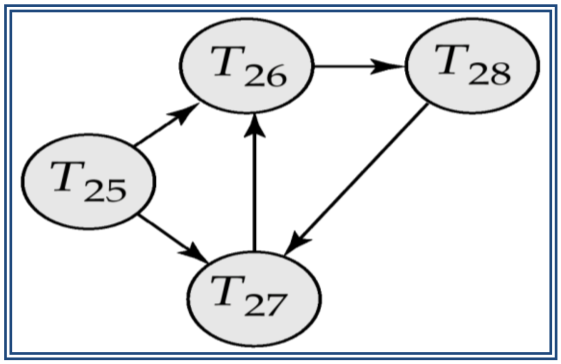
Deadlock *Detection*

- **Deadlocks** can be described as a *wait-for graph*,
  - which consists of a pair $G = (V,E)$,
    - $\boldsymbol{V}$ is a set of vertices (all the transactions in the system)
    - $\boldsymbol{E}$ is a set of edges; each element is an ordered pair $\boldsymbol{T_i \rightarrow T_j}$.
- If $\boldsymbol{T_i \rightarrow T_j}$ is in E, then there is a directed edge from T_i to T_j,
  - implying that $\boldsymbol{T_i}$ is waiting for $\boldsymbol{T_j}$ to release a data item.
- When T_i requests a data item currently being held by T_j,
  - then the edge $\boldsymbol{T_i \rightarrow T_j}$ is inserted in the wait-for graph.
  - This edge is removed only when:
    - $\boldsymbol{T_j}$ is no longer holding a ***data item*** needed by $\boldsymbol{T_i}$.

- The system is in a **deadlock** state:
  - ***if** and only **if*** the wait-for graph **has a cycle**.
- **Must invoke** a **deadlock-detection** algorithm periodically:
  - to look for **cycles**.



Wait-for graph **without** a cycle



Wait-for graph **with** a cycle

Deadlock *Recovery*

- When deadlock is detected :
  - Some transaction will **have to** be **rolled back** (made a victim):
    - to **break** deadlock.
    - **Select** that transaction as victim that:
      - will **incur minimum** cost.
  - Rollback -- determine how far to roll back transaction
    - **Total** rollback:
      - **Abort** the transaction and then **restart** it.
    - More effective to roll back transaction **only**:
      - **as far as** necessary to **break** deadlock.
  - Starvation happens if **same** transaction is:
    - always chosen as **victim**.
    - **Include** the *number of rollbacks* in:
      - the **cost factor** to **avoid** starvation

Insert and Delete: *phantom* phenomenon

- If **two-phase** locking is used :
  - A **delete** operation may be performed only if the transaction deleting the tuple has an **X-lock** on the tuple to be deleted.
  - A transaction that **inserts** a new tuple into the database is given an **X-lock** on the tuple
- **Insertions** and **deletions can lead to** the **phantom phenomenon**:

- **T_i scans** a relation (e.g., **find all** accounts in Perryridge) and
- **T_j inserts** a tuple in the relation (e.g., insert a new account at Perryridge)
    - may conflict in spite of **not accessing** any tuple in common.
- **If** only **tuple locks** are used:
    - **non-serializable** schedules **can result**:
    - the **scan** transaction **may not see** the new account,
    - (yet may be serialized before the insert transaction).

- The transaction **scanning** the relation is reading *information* that:
    - indicates *what tuples* the relation contains,
  - while a transaction **inserting** a tuple updates the ***same*** *information*.
  - This *information* **should** be ***locked***.
- One **solution**:
  - **Associate** a *data item* with the relation,
    - to **represent** the *information* about what tuples the relation contains.
  - Transactions **scanning** the relation:
    - **acquire** a ***shared lock*** in the *data item*,
  - Transactions **inserting** or **deleting** a tuple:
    - **acquire** an ***exclusive lock*** on the *data item*.
    - (Note: locks on the *data item* do not conflict with locks on individual tuples.)
- Above protocol provides very **low** concurrency for insertions/deletions.
- **Index locking** protocols provide higher **concurrency**
  - While **preventing** the phantom phenomenon,
  - by **requiring** *locks* on certain *index buckets*.

Index Locking **Protocol**

- Every relation **must have at least** one index.
- **Access** to a relation **must be** made:
  - **only** through **one** of the indices on the relation.
- A transaction T_i that performs a lookup:
  - **must lock all** the *index buckets* that it accesses, in S-mode.
- A transaction T_i **may not** insert a tuple t_i into a relation r
  - without updating **all** indices to r.
- T_i must perform a lookup on **every** index to find:
  - ***all*** *index buckets* that could have possibly contained a pointer to tuple t_i,

      - had it existed already, and
  - **obtain** locks in **X-mode** on *all these index buckets*.
  - T_i **must** also **obtain** locks in **X-mode** on *all index buckets* that it **modifies**.
- The rules of the **two-phase** locking protocol must be observed
  - **Guarantees** that ***phantom*** *phenomenon* **won't occur!**

Weak Levels of Consistency

- **Degree-two *consistency*:**
  - differs from **two-phase** locking in that:
    - **S-locks** *may be released* at any time, and
    - **locks** *may be **acquired*** at any time
  - **X-locks** *must be held* **till** end of transaction
  - Serializability is **not *guaranteed***,
    - ***programmer*** must ensure that **no *erroneous*** database state will **occur!**
- **Cursor stability (CS)**:
  - *Special case* of ***degree-two*** *consistency*
  - For **reads**, each tuple is:
    - locked,
    - read, and
    - [lock is immediately released]
  - **X-locks** are *held* **till** end of transaction

Weak Levels of Consistency in SQL

- **SQL allows non-serializable** executions:
  - **Serializable:** is the default
  - **Repeatable read**:
    - **allows** *only **committed** records* to be read, and
    - **repeating** a read should return the same value
      - (so read locks should be retained)
    - However, the ***phantom*** phenomenon *need **not** be **prevented***
      - $\mathbf{T_1}$ may see some **records** *inserted* by $\mathbf{T_2}$,
      - but may not see **others** *inserted* by $\mathbf{T_2}$
  - **Read committed**:
    - same as **degree-two** consistency,
    - but ***most systems*** implement it as cursor-stability
  - **Read uncommitted**:
    - **allows** even **uncommitted data** to be **read**

Concurrency in *Index* Structures

- **Indices** are **unlike** other database items in that:
    - their **only** job is to **help** in **accessing data**.
- **Index-structures** are typically **accessed very often**,
    - *much more than* **other** database **items**.
- **Treating** *index-structures* **like** *other database items* leads to:
    - **low *concurrency***.
    - ***Two-phase locking*** on an **index** may result in:
        - transactions **executing** practically one-at-a-time!
- It is **acceptable** to **have nonserializable** concurrent **access** to an index:
    - as long as the **accuracy** of the index is **maintained**.
- In particular, the exact values read in an **internal node** of a **B^+-tree** are **irrelevant** so long as **we land up** in the **correct** leaf node.
- There are *index concurrency protocols* where:
    - **locks** on **internal nodes** are **released** early,
        - and **not** in a two-phase fashion.

Example of *index concurrency protocol*:

- Use **crabbing** instead of **two-phase** locking on the nodes of the **B^+-tree**, as follows.
- During search/insertion/deletion:
    - First **lock** the **root** node in shared mode.
    - After **locking all** *required* **children** of **a node** in shared mode,
        - release the **lock** on **the node**.
    - During insertion/deletion,
        - **upgrade** leaf node **locks** to exclusive mode.
    - When splitting or coalescing requires changes to a **parent**,
        - **lock** the **parent** in exclusive mode.
- Above protocol can cause excessive **deadlocks**.
    - Better protocols are available;
        - E.g the **B-link tree** protocol

RECOVERY SYSTEM

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Shadow Paging
- Recovery With Concurrent Transactions

- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Advanced Recovery Techniques
- ARIES Recovery Algorithm
- Remote Backup Systems

Failure *Classification*

- **Transaction failure** :
    - **Logical errors**: transaction cannot complete due to some internal error condition
    - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a *power* failure or other *hardware* or *software* failure causes the system to crash.
    - **Fail-stop assumption**: non-volatile ***storage contents*** are assumed to ***not be corrupted*** by system crash
        - Database systems have *numerous integrity checks* to ***prevent corruption*** of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
    - **Destruction** is assumed to be **detectable**: disk drives use checksums to detect failures

Recovery Algorithms

- Recovery algorithms are ***techniques*** to **ensure** database **consistency** and transaction **atomicity** and **durability** despite failures
- Recovery algorithms have ***two parts:***
    - Actions taken during normal transaction processing to **ensure** enough information **exists** to recover from failures
    - Actions taken after a failure to **recover** the database contents to a **state** that **ensures** atomicity, consistency and durability

Storage Structure

- **Volatile storage**:
    - does not survive system crashes
    - Ex: main memory, cache memory
- **Nonvolatile storage**:
    - survives system crashes

  - Ex: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
- **Stable storage**:
  - a *mythical* form of storage that survives **all failures**
  - *approximated by* maintaining multiple copies on distinct nonvolatile media

***Stable-Storage* Implementation**

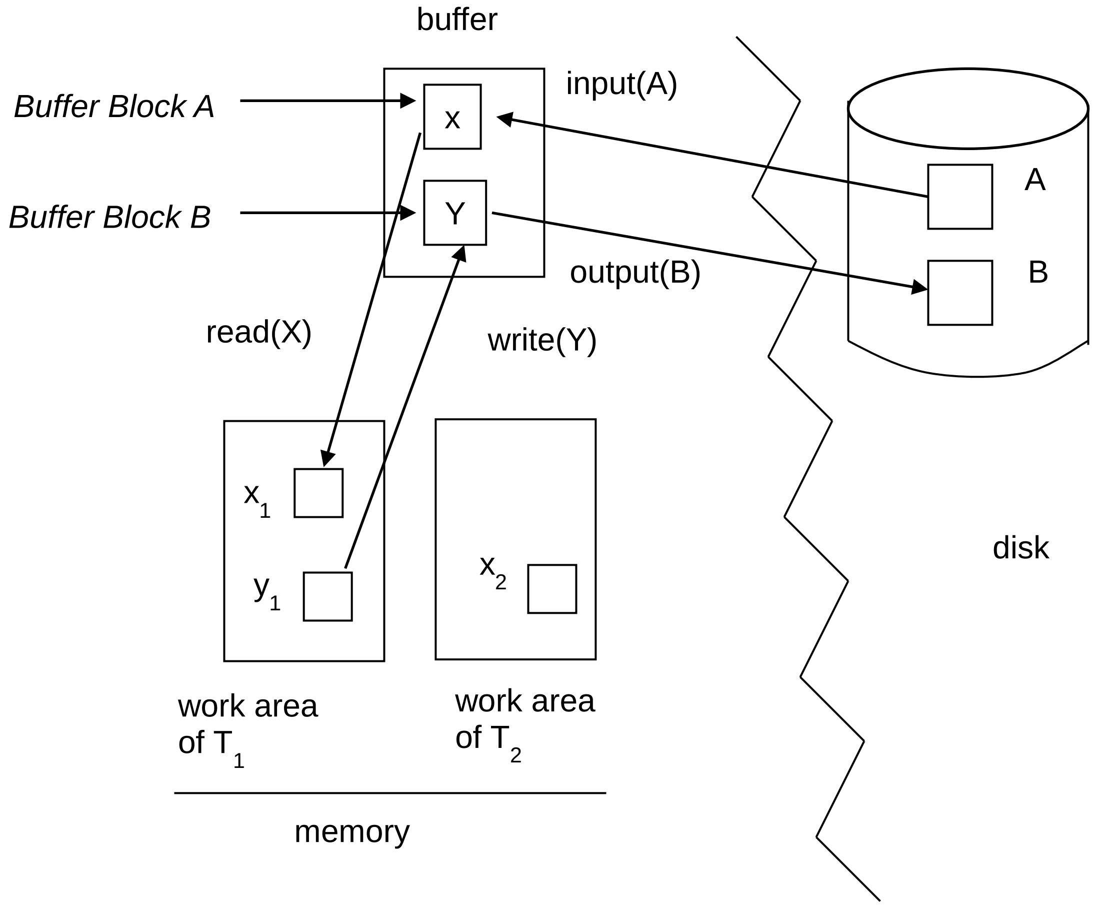
- Maintain **multiple** copies of each block on **separate** disks
  - copies can be at **remote** sites to *protect against disasters* such as **fire** or **flooding**.
- Failure during data transfer can still result in **inconsistent copies**:
  - Block transfer can result in:
    - Successful completion
    - Partial **failure**: destination block has incorrect information
    - Total **failure**: destination block was never updated
- Protecting storage media from **failure** during data transfer (one solution):
  - Execute output operation as follows (assuming **two copies** of each block):
    - ***Write*** the information onto the first physical block.
    - ***When*** the first write successfully completes, ***write*** the same information onto the second physical block.
    - The output is completed only after the second write successfully completes.
- Copies of a block **may differ** due to failure during output operation.
  To **recover** from failure:

1. First **find** inconsistent blocks:
    1. *Expensive solution*: **Compare** the two copies of every disk block.
    2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
2. **If** either copy of an inconsistent block is **detected** to have an error (bad checksum), **overwrite** it by the other copy. **If** both have **no error**, but are different, overwrite the second block by the first block.

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block **movements** between *disk* and *main* memory are initiated through the following two operations:
    - **input**(*B*) transfers the ***physical block B*** to main memory.
    - **output**(*B*) transfers the ***buffer block B*** to the disk, and replaces the appropriate physical block there.
- Each transaction T_i has its **private** *work-area* in which *local copies* of **all** data items accessed and updated by it are kept.
    - T_i's local copy of a data item ***X*** is called x_i.
- We assume, for *simplicity*, that each data item **fits** in, and is stored inside, a single block.
- Transaction ***transfers*** data items between ***system*** buffer blocks and its *private work-area* using the following operations :
    - **read**(*X*) assigns the value of data item ***X*** to the *local variable* x_i.
    - **write**(*X*) assigns the value of local variable x_i to data item {***X***} in the buffer block.
    - both these *commands* may ***necessitate*** the issue of an **input**(B_X) instruction before the assignment, if the block B_X in which ***X*** resides is not already in memory.
- Transactions
    - Perform **read**(*X*) while accessing *X* for the first time;
    - All subsequent accesses are to the ***local*** copy.
    - After last access, transaction executes **write**(*X*).
- **output**(B_X) need **not** immediately follow **write**(*X*).
    - *System* can perform the **output** operation *when it deems fit*.

Example of Data Access



Recovery and *Atomicity*

- **Modifying** the database **without** ensuring that the **transaction** will **commit**
    - may **leave** the database in an inconsistent state.
- Consider transaction T_i that transfers $50 from account *A* to account *B*; goal is:
    - **either** to perform **all** database modifications made by T_i
    - **or none** at all.
- **Several output** operations may be required for T_i (to output *A* and *B*).
    - A failure may occur **after one** of these *modifications* have been made
    - but before all of them are made.
- To **ensure** atomicity despite failures,
    - we first **output** information **describing** the modifications to stable storage without *modifying* the *database* itself.

- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that *transactions run **serially***, that is, one after the other.

Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and
    - maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a <T_i **start**> log record
- *Before* T_i executes **write**(X), a log record <T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j:
    - X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it's *last statement*, the log record **<T_i commit>** is written.
- We *assume* for now that log records are written *directly* to *stable storage*
  - (that is, they are ***not buffered***)
- Two approaches using logs:
  - **Deferred** database modification
  - **Immediate** database modification

Deferred Database Modification

- The **deferred** database modification scheme records all modifications to the log,
  - but defers **all** the **write**s to *after partial **commit***.
- Assume that *transactions execute **serially***
- Transaction starts by writing **<T_i *start*>** record to log.
- A **write**(X) operation results in a log record **<T_i, X, V>** being written, where V is the new value for X
  - Note: ***old value*** is not needed for this scheme
- The **write** is **not performed** on X at this time, but is deferred.
- When T_i partially commits, **<T_i commit>** is written to the log
- Finally, the log records are ***read** and **used*** to:
  - actually **execute** the previously deferred **writes**.
- During ***recovery*** after a crash, a transaction needs to be **redone**:

- *if and only if* **both** <T_i **start**> and <T_i **commit**> **are** there in the log.

- Redoing a transaction $\boldsymbol{T_i}$ (**redo** $\boldsymbol{T_i}$) sets the value of **all data** items:
  - updated by the transaction to the new values.
- Crashes can occur while:
  - the transaction is executing the original updates,
  - **or:** while ***recovery action*** is being taken
- Ex: transactions T_0 and T_1 ($\boldsymbol{T_0}$ executes before $\boldsymbol{T_1}$):

  T_0: **read** (A)
  - *A: - A - 50*
  - **Write** (A)
  - **read** (B)
  - *B:- B + 50*
  - **write** (B)

  T_1 : **read** (C)
  - *C:- C- 100*
  - **write** (C)
- Below we show the log as it appears at three instances of time:

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

- If log on stable storage at **time of crash** is as in case:
  - (a) No redo actions need to be taken
  - (b) **redo(*T*$_0$)** must be performed since <T_0 **commi**t> is present
  - (c) **redo**(T_0) must be performed followed by **redo(*T*$_1$)** since <T_0 **commit**> and <T_i commit> are present

Immediate Database Modification

- The **immediate database modification** scheme allows:
  - database ***updates*** of an ***uncommitted*** *transaction* to be made:
    - ***as*** *the writes are issued*
  - since, undoing may be needed,
    - update **logs** must have **both** old value and new value
- Update log record ***must*** *be written* ***before*** database *item is written*
  - We assume that the **log** record is *output directly* to stable storage
    - Can be extended to **postpone** log record output,

- Prior to execution of an **output**(B) operation for a *data block B*,
  - **all** log records corresponding to items B **must be flushed** to stable storage
- Output of updated blocks can take place at:
  - ***any time*** before or after transaction **commit**
- Order in which blocks are output ***can be different*** from:
  - the order in which they are written.

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
T_0, B, 2000, 2050		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.

- **Recovery** procedure has ***two operations*** instead of one:
  - **undo(T_i)** restores the value of all data items updated by T_i to their old values, ***going backwards*** from the last log record for T_i
  - **redo(T_i)** sets the value of all data items updated by T_i to the new values, ***going forward*** from the first log record for T_i
- **Both** operations must be **idempotent**
  - That is, even if the operation is executed ***multiple times*** the effect is the same as if it is executed once
    - Needed since operations may get ***re-executed*** during recovery
- When recovering after failure:

- Transaction T_i needs to be undone if the log contains the record <T_i **start**>,
  - but does not contain the record <T_i **commit**>.
- Transaction T_i needs to be redone if the log contains:
  - both the record <T_i **start**> and the record <T_i **commit**>.
- **Undo** operations are **performed first**, then **redo** operations.

***Immediate* DB Modification Recovery**
Below we show the log as it appears at three instances of time:

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:
(a) **undo (*T_0*)**: B is restored to 2000 and A to 1000.
(b) **undo (*T_1*)** and **redo (*T_0*)**: C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.
(c) **redo (*T_0*)** and **redo (*T_1*)**: A and B are set to 950 and 2050 respectively. Then *C* is set to 600

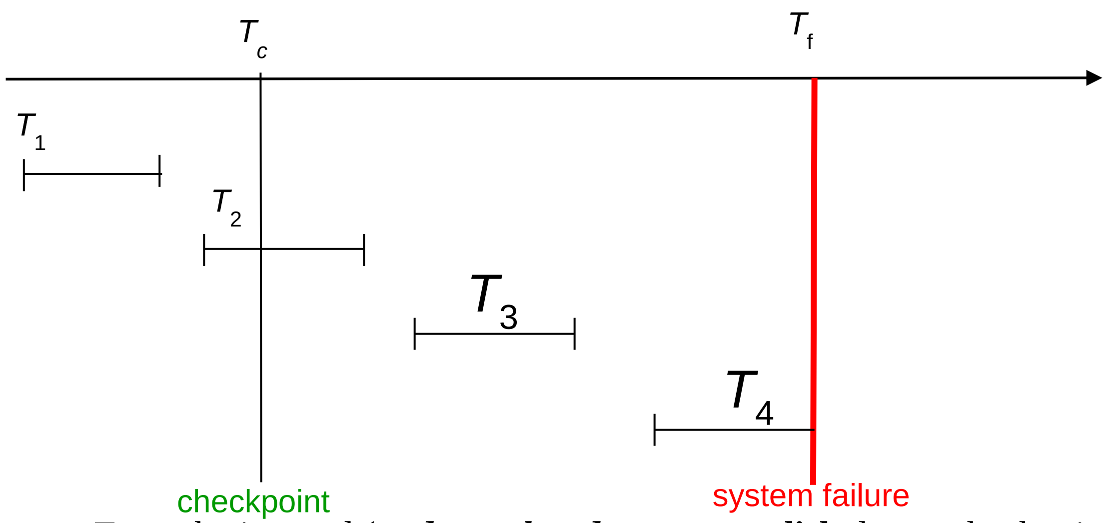
Checkpoints

- **Problems** in recovery procedure **as** *discussed earlier* :
  1. searching the entire log is **time-consuming**
  2. we might ***unnecessarily redo*** transactions which have ***already output*** their updates to the database.
- **Streamline recovery** procedure by periodically performing **checkpointing**
  1. Output **all log records** currently residing in main memory onto stable storage.
  2. Output **all** modified buffer blocks to the disk.

3. Write a log record < **checkpoint**> onto stable storage.

- During recovery we need to **consider** only the most recent transaction $\mathbf{T_i}$ that started before the checkpoint, and transactions that started after $\mathbf{T_i}$.
    1. Scan **backwards** from end of log to find the most recent <**checkpoint**> record
    2. Continue scanning **backwards** till a record <$\mathbf{T_i}$ **start**> is found.
    3. Need only **consider** the part of log **following** above **star**t record. *Earlier part of log* can be ignored during recovery, and can be erased whenever desired.
    4. For **all transactions** (starting from $\mathbf{T_i}$ or later) with no <$\mathbf{T_i}$ **commit**>, execute **undo($\mathbf{T_i}$)**. (Done only in case of immediate modification.)
    5. Scanning **forward** in the log, for **all transactions** starting from $\mathbf{T_i}$ or later with a <$\mathbf{T_i}$ **commit**>, execute **redo($\mathbf{T_i}$)**.

Example of Checkpoints



- T_1 can be ignored (***updates already output*** to **disk** due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

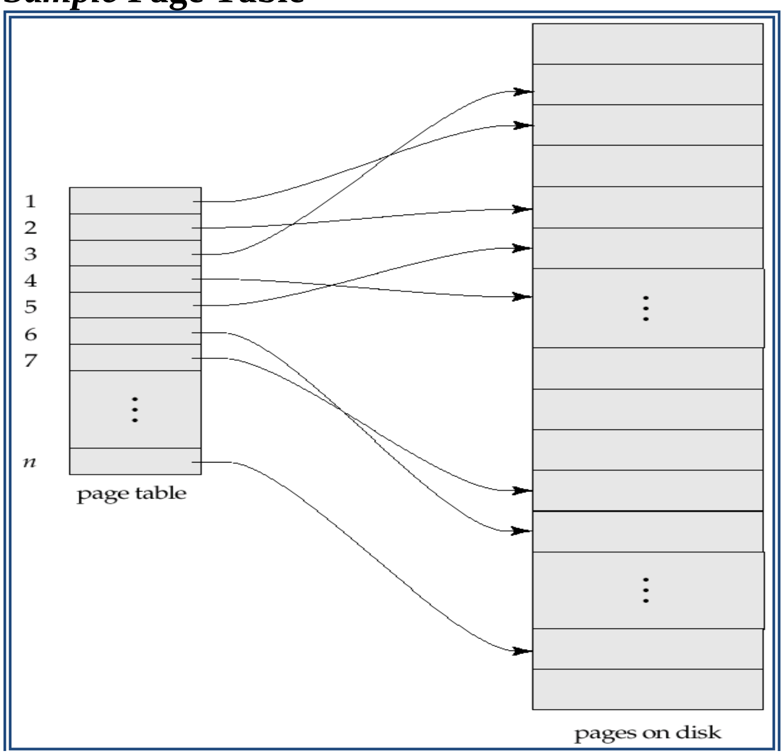
Shadow Paging

- **Shadow paging** is an **alternative** to ***log-based*** *recovery*;
    - this scheme is useful **if** *transactions execute* ***serially***
- Idea: maintain ***two*** page tables during the lifetime of a transaction:
    - the **current** page table, and the ***shadow*** page table
- Store the ***shadow*** page table in nonvolatile storage,
    - such that state of the database prior to transaction execution may be recovered.

- *Shadow* page table is **never *modified*** during execution

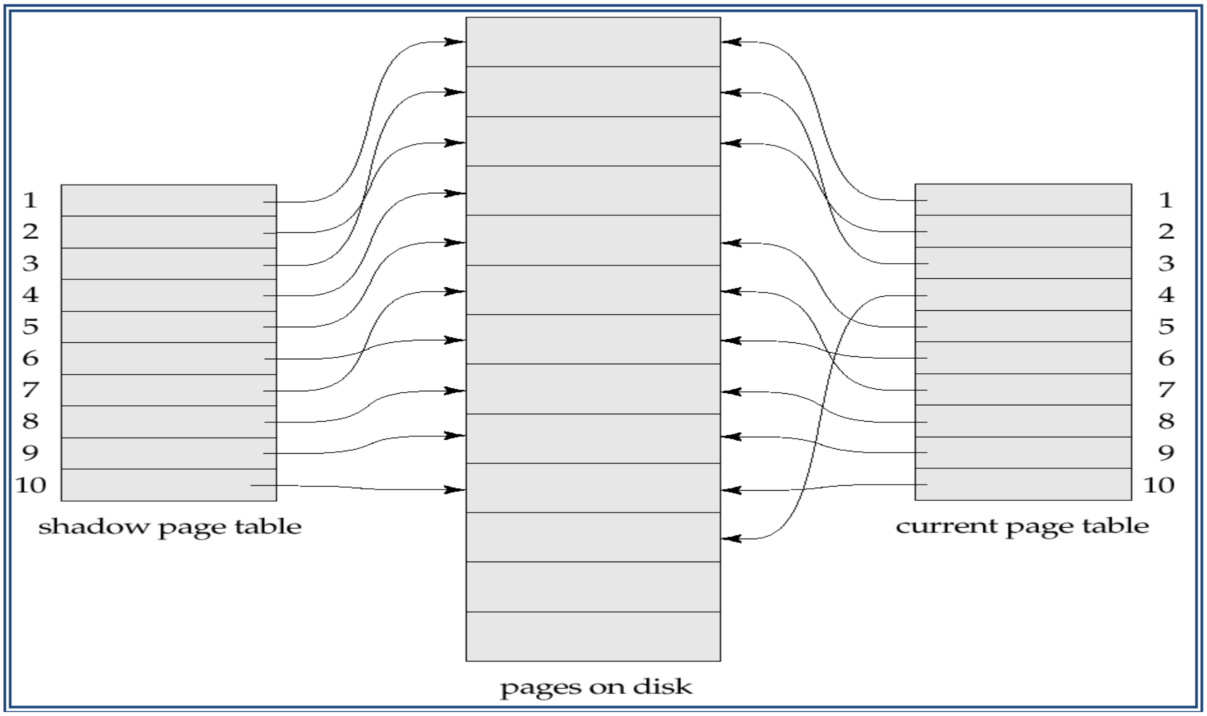
- To start with, **both** the page tables are **identical**.
    - **Only current** page table is used for:
        - data item ***accesses*** during execution of the transaction.
- Whenever **any page** is about to be written for the first time
    - **A copy** of this page is **made** onto an unused page.
    - The **current** page table is then **made** to point to the copy
    - The update is **performed** on the copy

Sample **Page Table**



Example of *Shadow* Paging

Shadow and **current** page tables after *write* to page **4**



- To **commit** a transaction :

1. Flush **all** *modified* pages in main memory to ***disk***
2. Output **current** page table to disk
3. Make the **current** page table the new ***shadow*** page table, as follows:
    - keep a ***pointer*** to the ***shadow*** page table at a fixed (known) location on ***disk***.
    - to make the **current** page table the new ***shadow*** page table,
        - *simply* **update** the ***pointer*** to point to:
            - **current** page table on ***disk***

- **Once pointer** to ***shadow*** page table has been *written*,
    - transaction is **committed**.
- **No** recovery is **needed** after a crash:
    - new transactions can **start** right away, using the ***shadow*** page table.
- **Pages** *not pointed* to from **current**/***shadow*** page table:
    - should be ***freed*** (*garbage collected*).
- Advantages of *shadow-paging* over *log-based* schemes
    - **no** overhead of ***writing log records***
    - recovery is **trivial**

- Disadvantages :
    - Copying the entire page table is very **expensive**
        - Can be reduced by using a page table structured like a B^+-tree
            - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
    - Commit **overhead** is high even with above extension
        - Need to **flush** every updated page, and page table
    - Data gets **fragmented** (related pages get separated on disk)
    - After every transaction completion, the database pages containing old versions of modified data need to be **garbage** collected
    - **Hard** to extend algorithm to **allow** transactions to run **concurrently**
        - Easier to extend log based schemes

Recovery With *Concurrent* Transactions

- We **modify** the ***log-based*** recovery schemes to:
        - allow **multiple** transactions to execute **concurrently**.
    - **All** transactions share a **single** disk buffer and a **single** log
    - A buffer block **can** have data items updated by **one** or **more** transactions
- We assume *concurrency control* using **strict** *two-phase locking*;
    - i.e. the **updates** of uncommitted transactions:
        - **should not** *be visible* to other transactions
        - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
- Logging is done as described *earlier*.
    - Log records of different transactions ***may*** *be* ***interspersed*** in the log.
- The checkpointing technique and ***actions*** taken on ***recovery:***
        - have to be changed**!**
    - since, ***several transactions*** *may be active*,
        - **when** a checkpoint is performed.

1. **Checkpoints** are performed as before,
        - except that the *checkpoint log record* is now of the form:

          < checkpoint *L*>

          where ***L*** is the list of transactions active at the time of the checkpoint
    - We assume **no** updates are in progress ,
        - **while** the checkpoint is carried out

- When the system **recovers** from a crash, it first does the following:

1. **Initialize** *undo-list* and *redo-list* to empty
2. **Scan** the log **backwards** from the end,
    - stopping when the first <**checkpoint** *L*> record is found.

For each record found **during** the *backward scan*:

- if the record is <T_i **commit**>, **add** T_i to *redo-list*
- if the record is <T_i **start**>, then **if** T_i is not in *redo-list*, **add** T_i to *undo-list*

3. For **every** $\boldsymbol{T_i}$ in ***L***, if $\mathbf{T_i}$ is not in *redo-list*, **add** $\boldsymbol{T_i}$ to *undo-list*

- At this point *undo-list* consists of *incomplete transactions* which must be **undone**, and *redo-list* consists of *finished transactions* that must be **redone**.
- Recovery now continues as follows:

4. **Scan** again the log **backwards** from the end**:**
    - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
    - Stop the scan when **<*T_i* start>** records have been *encountered* for **all** $\boldsymbol{T_i}$ in *undo-list*
5. **Locate** the most recent **<checkpoint *L*>** record.
6. **Scan** the log **forwards** from the <**checkpoint *L***> record**:**
    - During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*
    - Stop the scan at the *end of the log*.

Example of Recovery

- Go over the steps of the recovery algorithm on the following log:

<T_0 **star**t>
<T_0, *A*, 0, 10>
<$\boldsymbol{T_0}$ **commit**>
<$\boldsymbol{T_1}$ **start**> /* Scan in Step 4 stops here */
<T_1, *B*, 0, 10>
<$\boldsymbol{T_2}$ **start**>
<T_2, *C*, 0, 10>
<T_2, *C*, 10, 20>
<**checkpoint** {$\boldsymbol{T_1}$, $\boldsymbol{T_2}$}>
<T_3 **start**>
<T_3, *A*, 10, 20>
<T_3, *D*, 0, 10>
<$\boldsymbol{T_3}$ **commit**>

Log Record ***Buffering***

- Log record ***buffering***:
  - *Log records* are buffered in **main memory**,
    - instead of being ***output*** *directly to* stable storage.
  - *Log records* are ***output*** to stable storage:
    - when a block of log records in the ***buffer*** is **full**,
    - or a **log force** operation is executed.
- Log force is performed to **commit** a transaction by:
  - ***forcing*** all its log records (including the commit record) to stable storage.
- **Several** log records can thus be output using a ***single output*** operation,
  - **reducing** the **I/O** ***cost.***
- The **rules** below **must** *be followed* **if** *log records* are *buffered*:
  - Log records are ***output*** to stable storage:
    - in the **order** in which they are created.
  - Transaction T_i enters the **commit** state:
    - **only** when the *log record* <T_i **commit**> *has been* ***output*** to stable storage.
  - Before a **block** of data in main memory is ***output*** to the database,
    - **all** *log records* pertaining to data in that **block**:
      - **must** *have been* ***output*** to stable storage.
    - This rule is called the **write-ahead** logging or **WAL** rule:
      - Strictly speaking **WAL** only requires *undo information* to be ***output***

Database ***Buffering***

- Database maintains an **in-memory** *buffer* of *data blocks*
  - When a *new block* is needed, if **buffer** is **full**:
    - an existing ***block*** *needs to be removed* from buffer
  - **If** the ***block*** chosen for removal has been **updated**,
    - it **must** be output to disk
- As a result of the **write-ahead** logging rule,
  - if a ***block*** with *uncommitted* **updates** is ***output*** *to disk,*
    - *log records* with *undo information* for the updates are:
      - ***output*** *to the log* on stable storage **first**.
- **No** ***updates*** *should be* in progress on a **block**:
  - when it is **output** to disk.
  - Can be ensured *as follows*.

- Before **writing** a *data item,*
  - transaction acquires **exclusive lock** on **block** containing the data item
  - Lock can be **released** once the *write is completed*.
    - Such *locks* held for *short duration* are called **latches**.
- Before a **block** is ***output** to disk*,
  - the system acquires an **exclusive latch** on the **block**
  - **Ensures *no update*** *can* be *in progress* on the **block**
- Database **buffer** can be implemented:
  - either **in** an area of **real** main-memory **reserved *for*** *the database,*
  - or **in virtual** memory
- Implementing **buffer** in reserved main-memory has **drawbacks**:
  - Memory is ***partitioned before-hand*** between database buffer and applications,
    - ***limiting flexibility***.
  - **Needs may change**, and
    - although operating system **knows best**:
      - **how** memory *should be divided up* at **any time**,
      - **it *cannot change*** the partitioning of memory.
- Database **buffers** are:
  - **generally** *implemented* **in virtual** memory
    - in spite of some **drawbacks** *(as follows)***:**
- When operating system **needs** to **evict** a page that has been modified,
    - to make **space** for another page,
      - the page is written to **swap** space on **disk**.
- When database **decides** to **write** buffer page to **disk**,
  - buffer page *may be* in **swap** space, and
    - *may have* to be **read** from **swap** space on **disk** and
    - **output** to the database on disk,
    - resulting in **extra I/O!**
  - Known as ***dual paging*** *problem*.
  - **Ideally** when ***swapping*** *out* a **database** buffer page,
    - operating system *should pass control to* database,
      - which in turn **outputs** page to database (space)
      - instead of to **swap** space
    - (making sure to **output** log records **first**)
  - ***Dual paging*** can thus **be avoided**,
    - **but** common operating systems
      - do **not support** such *functionality*.

Failure with *Loss of Nonvolatile* Storage

- So far we assumed**:**
    - **no loss** of **non-volatile** storage
- ***Technique*** similar to checkpointing used
    - to deal with **loss** of **non-volatile** storage.
- Periodically **dump** the entire content of the database to **stable** storage
- No transaction may be **active** *during the dump* procedure;
    - a **procedure** similar to checkpointing **must** take place.
    - Output **all** log records:
        - currently residing in main memory onto stable storage.
    - Output **all** buffer blocks onto the **disk (***i.e., database***)**.
    - Copy the **contents** of the database to stable storage (*i.e., archival dump*).
    - Output a record <**dump**> to **log** on stable storage.
- To **recover** from ***disk failure***
    - restore **database** from most recent **dump**.
    - Consult the **log** and **redo all** transactions that committed after the **dump**
- Can be **extended** to allow **transactions** to be **active** during **dump**;
    - known as **fuzzy dump** or **online dump**
    - Will study *fuzzy checkpointing* later.

Advanced Recovery Algorithm

Advanced Recovery Techniques

- Support **high-concurrency** *locking techniques*,
    - such as those **used** for **B^+-tree** *concurrency control*
- Operations like **B^+-tree** insertions and deletions **release** locks **early**.
    - They **cannot** be **undone** *by restoring old values* (**physical undo**),
        - since once a lock is released,
            - other transactions may have updated the **B^+-tree**.
    - Instead, insertions (resp. deletions) are **undone** by:
        - executing a deletion (resp. insertion) operation (known as **logical undo**).

- For such operations, **undo** log records should contain:
    - the ***undo*** *operation* to be executed
    - called **logical undo** logging, in contrast to ***physical undo*** *logging*.
- **Redo** information is logged **physically**
    - (that is, new value for each write) even for such operations

        - Logical **redo** is very **complicated !**
            - since database state on **disk** may **not** be "*operation consistent*"
- Operation logging is done as follows:
    1. When operation starts, log **<*T_i*, *O_j*, operation-begin>**.
        - Here ***O_j*** is a **unique** identifier of the operation instance.
    2. While operation is executing,
        - normal log records with physical redo and physical undo *information* are logged.
    3. When operation completes, **<*T_i*, *O_j*, operation-end, *U*>** is logged,
        - where ***U*** contains *information* needed to perform a **logical** undo.

- If crash/rollback occurs **before** operation completes:
    - the **operation-end** log record is **not** found, and

    - the **physical** undo information is used to undo operation.
- If crash/rollback occurs **after** the operation completes:
    - the **operation-end** log record **is** found, and in this case
    - **logical** undo is performed using ***U***;

        - the **physical** undo *information* for the operation is *ignored*.
- **Redo** of operation (after **crash**):
    - still uses **physical** redo *information*.

Rollback of transaction T_i is done as follows:

- Scan the log backwards
    1. If a log record <T_i, X, V_1, V_2> is found,
        - perform the undo and
        - **log** a special **redo-only** log record <T_i, X, V_1>.
    2. If a <T_i, O_j, **operation-end**, ***U***> record is found
        - Rollback the operation **logically** using the undo information ***U***.
            - Updates performed during roll back **are logged**
                - » just like during *normal operation* execution.
            - At the end of the operation rollback,
                - » instead of logging an **operation-end** record,
                - » **generate** a record <T_i, O_j, **operation-abort**>.
        - **Skip** all preceding log records for T_i until:
            - the record <T_i, O_j **operation-begin**> is found
    3. **If** a **redo-only** record is found **ignore it**

    4. **If** a <T_i, O_j, **operation-abort**> record is found:
        - **skip all** preceding log records for T_i until :
            – the record <T_i, O_j, **operation-begi**n> is found.
    5. **Stop** the scan when the record <T_i, **start**> is found
    6. **Add** a <T_i, **abort**> record to the log

- Some points to note:
    - Cases 3 and 4 above can occur **only if**:
        - the database crashes while a transaction is being rolled back.
    - Skipping of log records as in case 4 is **important** :
        - to **prevent multiple** rollback of the *same operation*.

The following actions are taken when **recovering** from **system** crash**:**

1. Scan log forward from last < **checkpoint** *L*> record
    1. **Repeat history** by physically **redoing** :
        - **all** updates of **all** transactions,
    2. **Create** an undo-list during the scan as follows:
        - *undo-list* is set to ***L*** initially
        - Whenever <T_i **start**> is found T_i is added to *undo-list*
        - Whenever <T_i **commit**> or <T_i **abort**> is found, T_i is deleted from *undo-list*
    - This **brings database** to state as of crash,
        - with committed as well as uncommitted transactions having been **redone**.
    - Now *undo-list* contains transactions that are **incomplete**, that is,
        - have neither committed nor been **fully** rolled back.
2. **Scan** log backwards, performing undo on log records of transactions found in *undo-list*.
    - Transactions are rolled back
        - as described earlier.
    - When <T_i **start**> is found for a transaction T_i in *undo-list*,
        - write a <T_i **abort**> log record.
    - **Stop** scan when <T_i **start**> records have been found for **all** T_i in *undo-list*

- This **undoes** the effects of incomplete *transactions*
    - (those with neither **commit** nor **abort** log records).
- **Recovery** is now **complete**.

- **Checkpointing** is done as follows:
    1. Output **all** log records in memory to **stable storage**
    2. Output **all** modified buffer blocks to **disk**

3. Output a < **checkpoint** *L*> record to log on **stable storage** .
4. Transactions are **not** allowed to perform **any** actions
   - while **checkpointing** is in progress.

- **Fuzzy** checkpointing **allows** *transactions* to **progress**
  - while the most time consuming **parts** of checkpointing are in progress

- **Fuzzy checkpointing** is done as follows:
  1. Temporarily **stop all** updates by transactions
  2. Write a <**checkpoint** *L*> log record and **force** log to *stable storage*
  3. Note list ***M*** of modified buffer blocks
  4. Now permit transactions to **proceed** with their actions
  5. Output to disk **all** modified buffer blocks in list *M*
     - blocks should not be updated **until** being **output**
     - Follow **WAL**: **all** log records pertaining to a block must be **output** before the block is output
  6. Store a pointer to the **checkpoint** record
     - in a fixed position **last_checkpoint** on **disk**

- When **recovering** using a **fuzzy** checkpoint,
  - **start scan** from the **checkpoint** record pointed to by **last_checkpoint**
  - Log records **before last_checkpoint**
    - have their updates reflected in database on disk, and
    - **need not** be redone.
  - Incomplete checkpoints,
    - where **system** had crashed while performing checkpoint,
      - are handled **safely**

PARALLEL DATABASES

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Design of Parallel Systems

Introduction

- Parallel machines are becoming quite common and affordable
    - Prices of microprocessors, memory and disks have dropped sharply
    - Recent desktop computers feature multiple processors and this trend is projected to accelerate
- Databases are growing increasingly large
    - large volumes of transaction data are collected and stored for later analysis.
    - multimedia objects like images are increasingly stored in databases
- Large-scale parallel database systems increasingly used for:
    - storing large volumes of data
    - processing time-consuming decision-support queries
    - providing high throughput for transaction processing

Parallelism in Databases

- Data can be partitioned across multiple disks for parallel I/O.
- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
    - data can be partitioned and each processor can work independently on its own partition.
- Queries are expressed in high level language (SQL, translated to relational algebra)
    - makes parallelization easier.
- Different queries can be run in parallel with each other. Concurrency control takes care of conflicts.
- Thus, databases naturally lend themselves to parallelism.

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning
- the relations on multiple disks.

- Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk.
- Partitioning techniques (number of disks = n):

Round-robin:
Send the i^{th} tuple inserted in the relation to disk i mod n.

Hash partitioning:
  - Choose one or more attributes as the partitioning attributes.
  - Choose hash function h with range **0…n - 1**
  - Let i denote result of hash function h applied to the partitioning attribute value of a tuple. Send tuple to disk i.
- **Range partitioning:**
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector [$v_0, v_1, ..., v_{n-2}$] is chosen.
  - Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \le v_{i+1}$ go to disk $I + 1$. Tuples with $v < v_0$ go to disk **0** and tuples with $v \ge v_{n-2}$ go to disk n**-1**.

E.g., with a partitioning vector [5,11], a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will go to disk 1, while a tuple with value 20 will go to disk 2.

Comparison of Partitioning Techniques
- Evaluate how well partitioning techniques support the following types of data access:

1. Scanning the entire relation.
2. Locating a tuple associatively – **point queries**.
   - E.g., r.A = 25.
3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
   - E.g., $10 \le r.A < 25$.

Round robin:
- Advantages
  - Best suited for sequential scan of entire relation on each query.
  - All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.
- Disadvantages
  - Range queries are difficult to process

    - No clustering -- tuples are scattered across all disks

Hash partitioning:

- Good for sequential access
    - **Assuming** hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
    - Retrieval work is then well balanced between disks.
- Good for point queries on partitioning attribute
    - Can lookup single disk, leaving others available for answering other queries.
    - Index on partitioning attribute can be local to disk, making lookup and update more efficient
- No clustering, so difficult to answer range queries

Range partitioning:

- Provides data clustering by partitioning attribute value.
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- For range queries on partitioning attribute, one to a few disks may need to be accessed
    - Remaining disks are available for other queries.
    - Good if result tuples are from one to a few blocks.
    - If many blocks are to be fetched, they are still fetched from one to a few disks, and potential parallelism in disk access is wasted
        - Example of execution skew.

Partitioning a Relation across Disks

- If a relation contains only a few tuples which will fit into a single disk block, then assign the relation to a single disk.
- **Large relations** are preferably partitioned **across all** the available **disks**.
- If a relation consists of ***m*** disk blocks and there are ***n*** disks available in the system, then the relation should be allocated **min**(*m*,*n*) disks.

Handling of Skew

- The distribution of tuples to disks may be **skewed** — that is, some disks have many tuples, while others may have fewer tuples.
- **Types of skew:**
    1) **Attribute-value skew.**

- Some values appear in the partitioning attributes of **many tuples**; all the tuples with the same value for the partitioning attribute end up in the same partition.
- Can **occur** with range-partitioning and hash-partitioning.

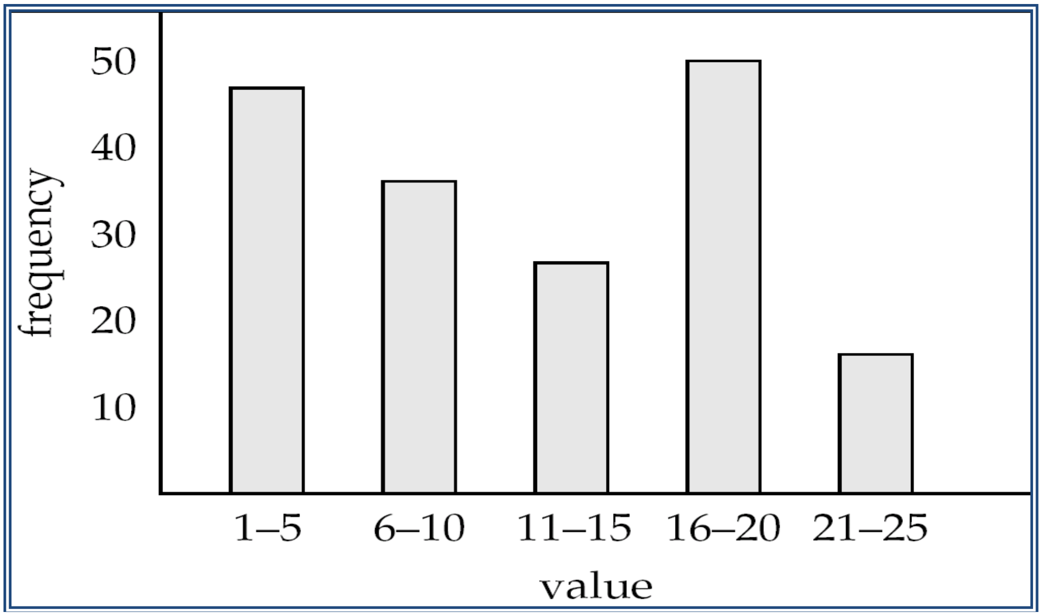
2) **Partition skew**.
    - With range-partitioning, **badly** chosen partition **vector** may assign too many tuples to some partitions and too few to others.
    - Less likely with hash-partitioning if a **good** hash-function is chosen.

Handling Skew in Range-Partitioning

- To create a **balanced** partitioning vector (assuming partitioning attribute forms a key of the relation):
    - **Sort** the relation on the partitioning attribute.
    - Construct the **partition vector** by scanning the relation in sorted order as follows.
        - After **every** $1/n^{th}$ of the relation has been read, the value of the partitioning attribute of the next tuple is **added to** the partition **vector**.
    - ***n*** denotes the **number** of **partitions** to be constructed.
    - Duplicate entries or imbalances can result **if duplicates** are present in partitioning attributes.
- Alternative technique based on **histograms** used in practice

Handling Skew using Histograms

- **Balanced** partitioning vector can be constructed from **histogram** in a relatively straightforward fashion
    - **Assume** uniform distribution within each range of the histogram
- Histogram can be constructed by scanning relation, or sampling (blocks containing) tuples of the relation



Handling Skew Using Virtual Processor Partitioning

- Skew in range partitioning can be handled **elegantly** using **virtual processor partitioning**:
  - create a large number of partitions (say **10** to **20** times the number of processors)
  - Assign virtual processors to partitions either in round-robin fashion or based on estimated cost of processing each virtual partition
- **Basic idea**:
  - If any normal partition would have been skewed, it is very likely the **skew** is **spread** over a number of virtual partitions
  - Skewed virtual partitions get **spread** across a number of **processors**, so work gets distributed evenly!

Interquery Parallelism

- **Queries**/transactions execute in **parallel** with one another.
- Increases transaction **throughput**; used primarily to **scale up** a transaction processing system to support a **larger number** of **transactions** per second.
- Easiest form of parallelism to support, particularly in a **shared-memory** parallel database, because even sequential database systems support concurrent processing.
- More **complicated** to implement **on** shared-disk or shared-nothing architectures:

- **Locking** and logging must be coordinated by passing messages between processors.
- Data in a **local buffer** may have been updated at another processor.
- **Cache-coherency** has to be **maintained** — reads and writes of data in buffer must find **latest version of data**.

Cache Coherency Protocol

- Example of a cache coherency **protocol** for shared disk systems:
  - Before **reading/writing** to a page, the **page** must be **locked** in shared/exclusive mode.
  - On locking a page, the page must be read from disk
  - Before **unlocking** a page, the **page** must be **written** to disk if it was modified.
- More **complex protocols** with fewer disk reads/writes **exist**.
- Cache coherency **protocols** for shared-nothing systems are similar. Each database page is assigned a ***home*** **processor**. Requests to fetch the page or write it to disk are sent to the home processor.

Intraquery Parallelism

- Execution of **a single query** in **parallel** on multiple processors/disks; important for **speeding up** long-running queries.
- **Two** complementary **forms** of intraquery parallelism :
  - **Intraoperation Parallelism** – parallelize the execution of **each** individual operation in the query.
  - **Interoperation Parallelism** – execute the **different** operations in a query expression in parallel.
- the **first** form **scales better** with increasing parallelism because the number of tuples processed by each operation **is** typically **more than** the number of operations in a query

Parallel Processing of Relational Operations

- Our discussion of parallel algorithms **assumes**:
  - *read-only* queries
  - shared-nothing architecture
  - ***n*** processors, $\boldsymbol{P_0, ..., P_{n-1}}$, and ***n*** disks $D_0, ..., D_{n-1}$, where disk D_i is associated with processor P_i.
- If a processor has multiple disks they can simply simulate a **single disk** D_i.

- Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems.
    - Algorithms for shared-nothing systems can thus be run on shared-memory and shared-disk systems.
    - However, some optimizations may be possible.

Parallel Sort

Range-Partitioning Sort

- Choose processors $\boldsymbol{P_0, ..., P_m}$, where $\boldsymbol{m \leq n - 1}$ to do sorting.
- Create range-partition vector with m entries, on the sorting attributes
- Redistribute the relation using range partitioning
    - all tuples that lie in the i^{th} range are sent to processor P_i
    - P_i stores the tuples it received temporarily on disk D_i.
    - This step requires **I/O** and **communication overhead**.
- Each processor $\boldsymbol{P_i}$ sorts its partition of the relation locally.
- Each processor executes same operation (sort) **in parallel** with other processors, without any interaction with the others (**data parallelism)**.
- Final **merge** operation is **trivial**: range-partitioning ensures that, for 1 j m, the key values in processor $\boldsymbol{P_i}$ are all less than the key values in $\boldsymbol{P_j}$.

Parallel External Sort-Merge

- Assume the relation has already been partitioned among disks $\boldsymbol{D_0, ..., D_{n-1}}$ (in whatever manner).
- Each processor $\boldsymbol{P_i}$ **locally** sorts the data on disk $\boldsymbol{D_i}$.
- The sorted runs on each processor are then merged to get the final sorted output.
- **Parallelize** the **merging** of sorted runs as follows:
    - The sorted partitions at each processor P_i are **range-partitioned** across the processors $\boldsymbol{P_0, ..., P_{m-1}}$.
    - Each processor $\boldsymbol{P_i}$ performs a merge on the streams as they are received, to get a single sorted run.
    - The sorted runs on processors $\boldsymbol{P_0,..., P_{m-1}}$ are concatenated to get the final result.

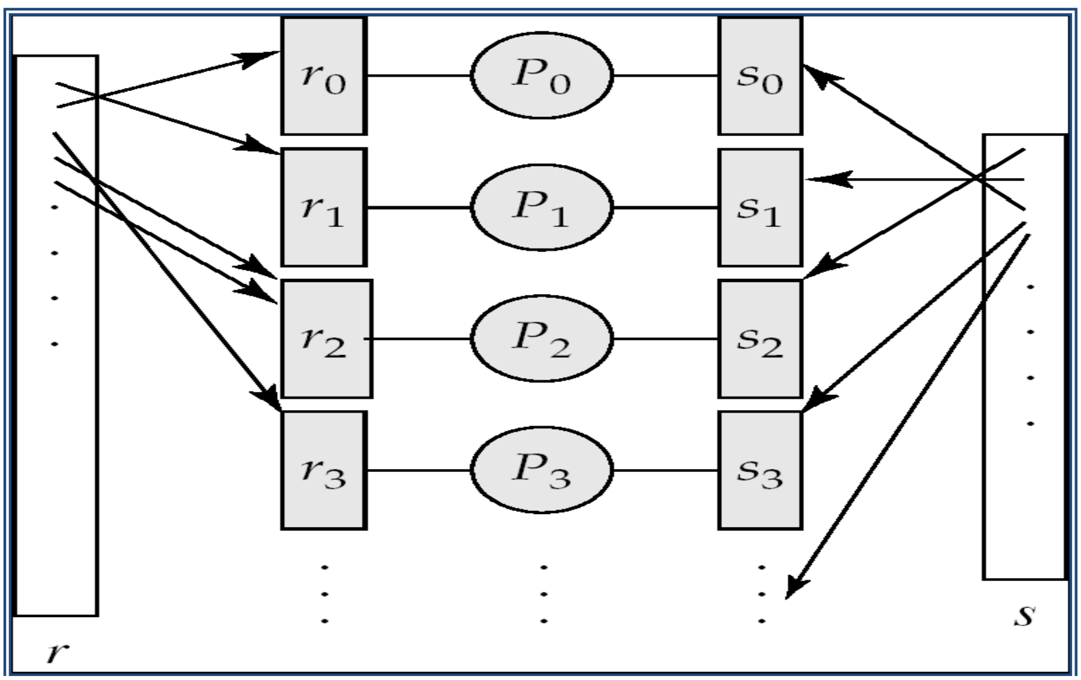
Parallel Join

- The join operation requires **pairs of tuples** to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to **split the pairs** to be tested **over several processors**. Each processor then computes part of the join locally.

- In a final step, the **results** from each processor can be **collected** together to produce the final result.

Partitioned Join

- For equi-joins and natural joins, it is possible to *partition* the two input relations across the processors, and compute the join locally at each processor.
- Let ***r*** and ***s*** be the input relations, and we want to compute $r \bowtie_{r.A = s.B} s$.
- ***r*** and ***s*** each are partitioned into ***n*** partitions, denoted $r_0, r_1, ..., r_{n-1}$ and $s_0, s_1, ..., s_{n-1}$.
- Can use either *range partitioning* or *hash partitioning*.
- ***r*** and ***s*** must be partitioned on their join attributes ***r*.A** and ***s*.B**), using the same range-partitioning vector or hash function.
- Partitions r_i and s_i are sent to processor P_i,
- Each processor P_i locally computes $r_i \bowtie_{ri.A = si.B} s_i$. Any of the standard join methods can be used.

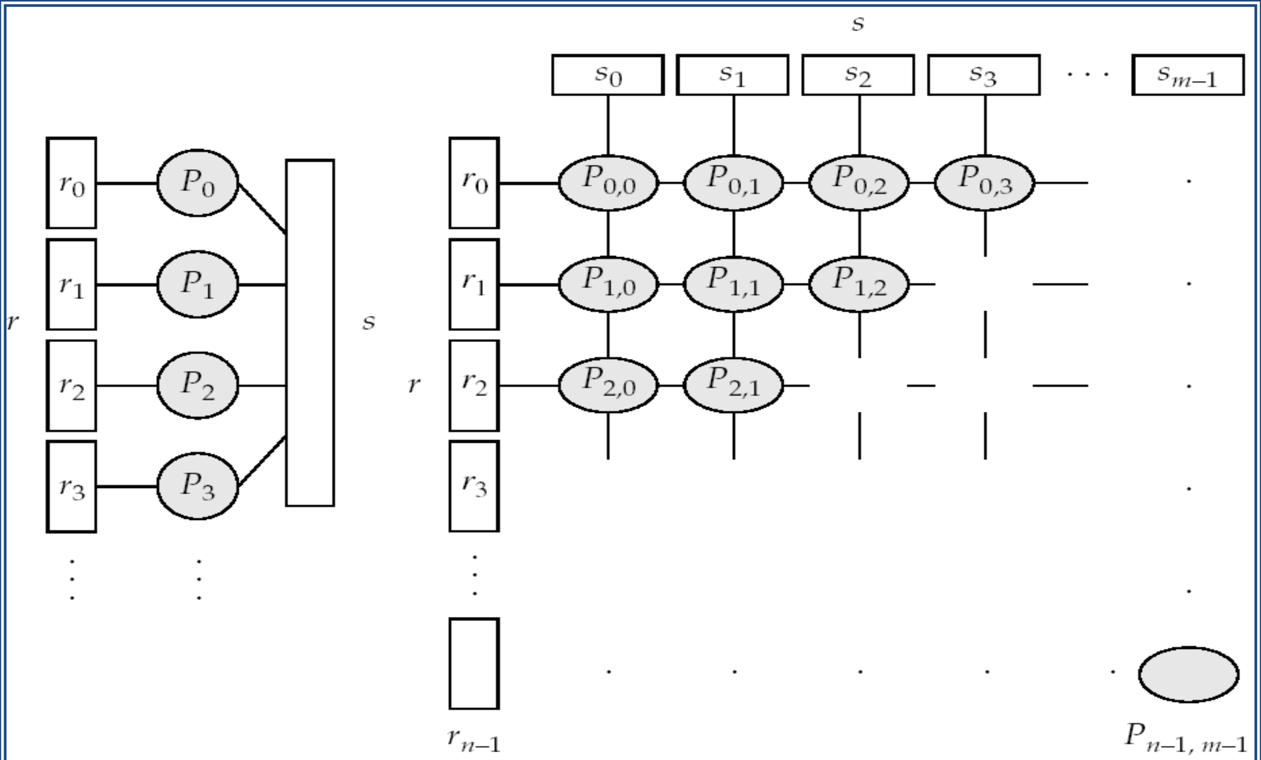


Fragment-and-Replicate Join

- Partitioning **not possible** for some join conditions
  - e.g., non-equijoin conditions, such as **r.A > s.B**.
- For joins were partitioning is not applicable, parallelization **can be** accomplished by **fragment and replicate** technique
- Special case – **asymmetric fragment-and-replicate**:

- One of the relations, say ***r***, is partitioned; any partitioning technique can be used.
- The other relation, ***s***, is replicated across all the processors.
- Processor $\boldsymbol{P_i}$ then locally computes the join of $\boldsymbol{r_i}$ with all of **s** using any join technique.

Depiction of Fragment-and-Replicate Joins



(a) Asymmetric fragment and replicate (b) Fragment and replicate

General case: reduces the sizes of the relations at each processor.

- ***r*** is partitioned into **n** partitions, r_0, r_1, ..., r_{n-1}; **s** is partitioned into ***m*** partitions, s_0, s_1, ..., s_{m-1}.
- Any partitioning technique may be used.
- There must be at least **m * n** processors.
- Label the processors as
- $P_{0,0}$, $P_{0,1}$, ..., $P_{0,m-1}$, $P_{1,0}$, ..., P_{n-1m-1}.
- $\boldsymbol{P_{i,j}}$ computes the join of $\boldsymbol{r_i}$ with $\boldsymbol{s_j}$. In order to do so, $\boldsymbol{r_i}$ is replicated to $P_{i,0}$, $P_{i,1}$, ..., $P_{i,m-1}$, while $\mathbf{s_i}$ is replicated to $P_{0,i}$, $P_{1,i}$, ..., $P_{n-1,i}$
- Any join technique can be used at each processor $\boldsymbol{P_{i,j}}$.

- Both versions of fragment-and-replicate **work** with any join condition, since **every** tuple in ***r*** can be **tested** with **every** tuple in ***s***.
- Usually has a **higher cost** than partitioning, since one of the relations (for asymmetric fragment-and-replicate) or both **relations** (for general fragment-and-replicate) have to be **replicated**.
- Sometimes **asymmetric** fragment-and-replicate is **preferable** even though partitioning could be used.
    - E.g., say ***s*** is **small** and ***r*** is **large**, and already partitioned. It may be **cheaper** to **replicate** ***s*** across all processors, rather than repartition ***r*** and ***s*** on the join attributes.

Partitioned Parallel Hash-Join

Parallelizing partitioned **hash join**:

- Assume ***s*** is **smaller** than ***r*** and therefore ***s*** is chosen as the build relation.
- A hash function h_1 takes the join attribute value of each tuple in ***s*** and maps this tuple to one of the ***n*** processors.
- Each processor P_i reads the tuples of ***s*** that are on its disk D_i, and **sends** each tuple to the appropriate processor based on **hash** function h_1. Let s_i denote the tuples of relation ***s*** that are sent to processor P_i.
- As **tuples** of relation ***s*** are **received** at the destination processors, they are **partitioned** further using another hash function, h_2, which is used to **compute** the hash-join **locally**.
- Once the tuples of ***s*** have been **distributed**, the larger relation ***r*** is redistributed across the ***m*** processors using the hash function h_1
- Let r_i denote the tuples of relation ***r*** that are sent to processor P_i.
- As the ***r*** tuples are received at the destination processors, they are repartitioned using the function h_2
- (just as the probe relation is partitioned in the sequential hash-join algorithm).
- Each processor P_i **executes** the build and probe **phases** of the hash-join algorithm on the local partitions r_i and ***s*** of ***r*** and ***s*** to produce a partition of the final result of the hash-join.
- Note: Hash-join **optimizations** can be applied to the **parallel** case
- e.g., the **hybrid** hash-join algorithm can be used to **cache** some of the **incoming tuples** in memory and avoid the cost of writing them and reading them back in.

Parallel Nested-Loop Join

- Assume that
  - relation *s* is much smaller than relation *r* and that *r* is stored by partitioning.
  - there is an index on a join attribute of relation *r* at each of the partitions of relation *r*.
- Use **asymmetric** fragment-and-replicate, with relation *s* being replicated, and using the existing partitioning of relation *r*.
- Each processor P_j where a partition of relation *s* is stored reads the tuples of relation *s* stored in D_j, and replicates the tuples to every other processor P_i.
  - At the end of this phase, relation *s* is replicated at all sites that store tuples of relation *r*.
- Each processor P_i performs an indexed nested-loop join of relation *s* with the i^{th} partition of relation *r*.

Other Relational Operations

Selection $\sigma_\theta(r)$:

- If θ is of the form $a_i = v$, where a_i is an attribute and v a value.
  - **If r** is **partitioned** on a_i the selection is performed at a single processor.
- If θ is of the form $l <= a_i <= u$ (i.e., θ is a range selection) and the relation has been **range-partitioned** on a_i
  - Selection is performed at each **processor** whose **partition overlaps** with the **specified range** of values.
- In all other cases: the **selection** is **performed** in parallel at **all** the **processors**.

- **Duplicate elimination:**
  - Perform by using **either of** the **parallel sort techniques**
    - **eliminate** duplicates **as soon as** they are **found** during sorting.
  - Can also **partition** the tuples (using either **range-** or **hash-**partitioning) and perform duplicate **elimination locally** at each processor.
- **Projection:**
  - Projection **without duplicate elimination** can be performed as tuples are read in from disk in parallel.
  - **If duplicate elimination** is **required**, any of the above duplicate elimination techniques can be used.

Grouping/Aggregation

- **Partition** the **relation on** the **grouping attributes** and then **compute** the aggregate values **locally** at each processor.
- Can **reduce cost of transferring tuples** during partitioning **by partly computing aggregate values** before partitioning.
- Consider the **sum** aggregation operation:
  - Perform aggregation operation at each processor $\mathbf{P_i}$ on those tuples stored on disk $\mathbf{D_i}$
    - **results in** tuples with **partial sums** at each processor.
  - Result of the **local aggregation** is partitioned on the **grouping attributes**, and the aggregation **performed again** at each processor $\mathbf{P_i}$ to **get** the **final result**.
- **Fewer tuples need to be sent** to other processors during partitioning.

Cost of Parallel Evaluation of Operations

- **If** there is **no skew** in the partitioning, **and** there is **no overhead** due to the parallel evaluation, expected **speed-up will be 1/n**
- **If skew** and **overheads** are also to be taken into account, the **time taken** by a parallel operation can be estimated as

  $\mathbf{T_{part} + T_{asm} + \max (T_0, T_1, \ldots, T_{n-1})}$
  - T_{part} is the time for partitioning the relations
  - T_{asm} is the time for assembling the results
  - T_i is the **time taken** for the operation at processor P_i
    - this needs to be estimated taking into account the **skew**, and the **time** wasted in **contentions**.

Interoperator Parallelism

- **Pipelined parallelism**
  - Consider a join of four relations
    - $\mathbf{r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4}$
  - Set up a pipeline that computes the three joins in parallel
    - Let **P1** be assigned the computation of

      $temp1 = r_1 \bowtie r_2$
    - And **P2** be assigned the computation of $temp2 = temp1 \bowtie r_3$
    - And **P3** be assigned the computation of $temp2 \bowtie r_4$
  - **Each** of these **operations** can **execute in parallel**, sending **result** tuples it computes to the next operation even as it is computing further results

- Provided a **pipelineable join** evaluation **algorithm** (e.g. indexed **nested loops join**) **is used**

Factors Limiting Utility of Pipeline Parallelism

- **Pipeline** parallelism **is useful** since it **avoids writing intermediate** results to disk
- Useful with **small number** of processors, **but does not scale up well** with more processors. One reason is that pipeline chains do not attain **sufficient length**.
- **Cannot pipeline operators** which do not produce output **until all inputs** have been accessed (**e.g**. aggregate and **sort**)
- **Little speedup** is obtained for the frequent cases of **skew** in which **one operator's** execution **cost** is much **higher** than the **others**

Independent Parallelism

- **Independent parallelism**
  - **Consider** a **join** of four relations

$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$

- Let $\mathbf{P_1}$ be assigned the computation of
  $temp_1 = r_1 \bowtie r_2$
- And P_2 be assigned the computation of $temp_2 = r_3 \bowtie r_4$
- And P_3 be assigned the computation of $temp_1 \bowtie temp_2$
- P_1 and P_2 can work independently **in parallel**
- P_3 has to wait for **input** from P_1 and P_2
  - **Can pipeline** output of P_1 and P_2 to P_3, combining independent parallelism and **pipelined parallelism**
- **Does not provide** a **high degree** of **parallelism**
  - useful with a lower degree of parallelism.
  - **less useful in** a **highly parallel system**,

Query Optimization

- Query **optimization in parallel databases** is significantly **more complex** than query optimization in sequential databases.
- **Cost models** are more **complicated**, since we must take into account partitioning **costs** and issues such as **skew** and resource **contention**.
- When **scheduling execution tree** in parallel system, must **decide**:
  - How to **parallelize** each **operation** and how many **processors** to use for it.

    - What operations to pipeline, **what operations** to execute independently in parallel, and what operations to execute sequentially, one after the other.
- **Determining** the amount of resources **to allocate** for each operation is a problem.
    - E.g., allocating more processors than optimal **can result** in high **communication overhead**.
- **Long pipelines** should be avoided as the final operation may wait a lot for inputs, while **holding** precious **resources**
- The **number of** parallel **evaluation plans** from which to choose from is **much larger** than the number of sequential evaluation plans.
    - Therefore **heuristics** are **needed** while optimization
- Two alternative **heuristics** for choosing parallel plans:
    - **No pipelining** and inter-operation pipelining; just parallelize **every operation** across all processors.
        - Finding best plan is now much easier --- use standard optimization technique, but with **new cost model**
        - **Volcano** parallel database popularize the **exchange-operator** model
            - exchange operator is introduced into **query plans** to partition and distribute tuples
            - each operation works independently on local data on each processor, in **parallel** with **other copies of** the **operation**
    - **First** choose most efficient sequential plan and **then** choose how best to parallelize the operations in that plan.
        - Can explore pipelined parallelism as an option
- Choosing a **good** physical organization (**partitioning** technique) is important to speed up queries.

Design of Parallel Systems

Some issues in the **design** of parallel systems:

- **Parallel loading** of **data** from external sources is needed in order to handle large volumes of incoming data.
- **Resilience** to **failure** of some processors or disks.
    - **Probability** of some disk or processor **failing** is **higher** in a parallel system.
    - **Operation** (perhaps with degraded performance) should be **possible** in spite of failure.

        - Redundancy achieved by **storing extra copy** of every data item at another processor.
- **On-line reorganization** of **data** and **schema** changes **must be supported**.
    - For example, **index** construction on **terabyte databases** can take hours or days even on a parallel system.
        - Need to **allow other processing** (insertions/deletions/updates) to be performed on relation even as index is being constructed.
    - Basic idea: index construction tracks changes and ``**catches up**'' on changes at the end.
- Also **need** support for **on-line** repartitioning and **schema** changes (executed **concurrently** with **other processing**).