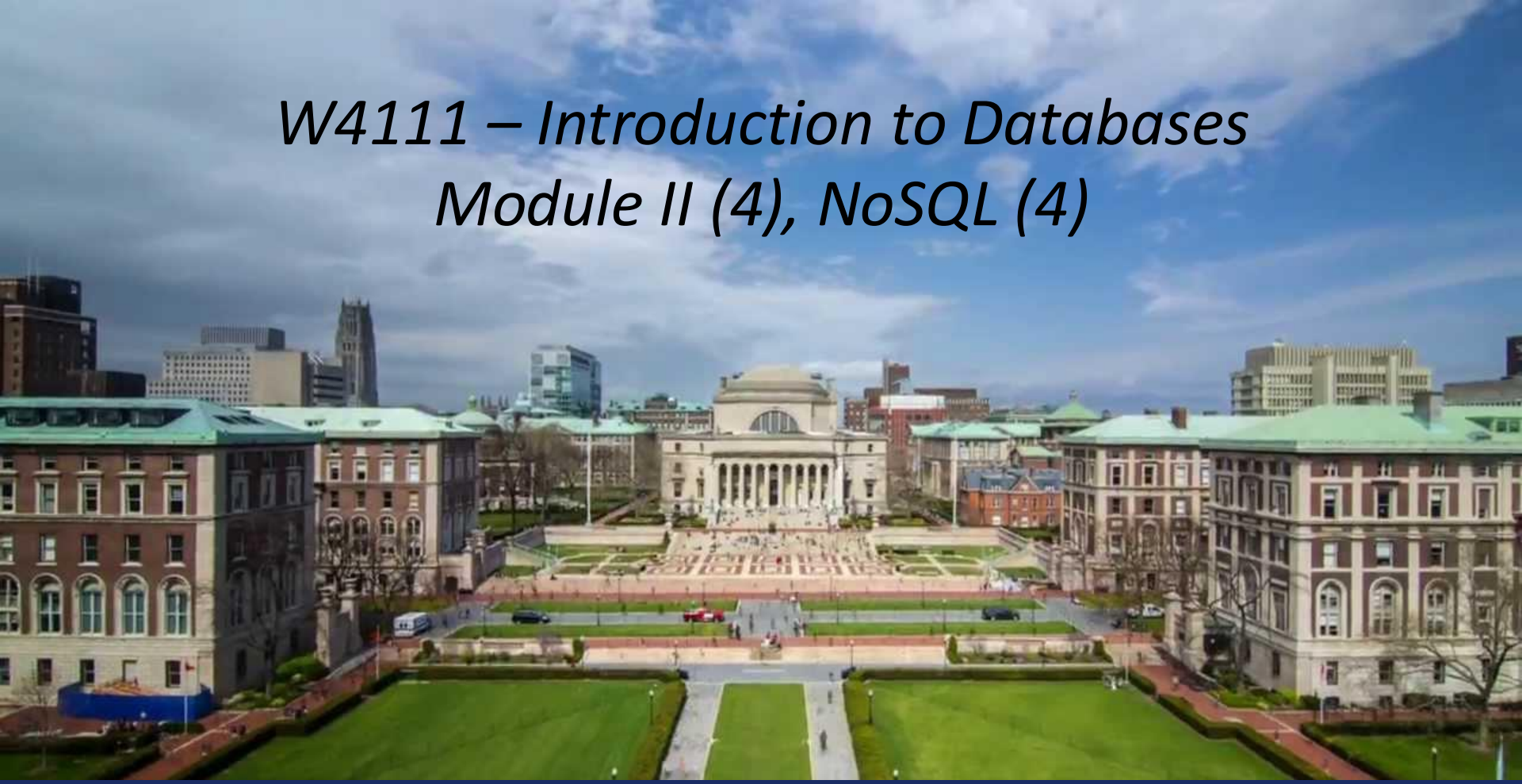
W4111 – Introduction to Databases
Module II (4), NoSQL (4)



COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Contents

Contents

- Course update
- Module II
  - Query optimization
    - Concepts
    - Equivalent expressions
    - Some examples of optimizations and explanation.
    - Statics and information for cost optimization computation
  - Transactions
    - Concepts
    - Atomicity, durability and recovery
    - Isolation, concurrency control and locking
  - Scalability
- Web applications
  - Concepts
  - REST
  - Walkthrough of project template

Course Update

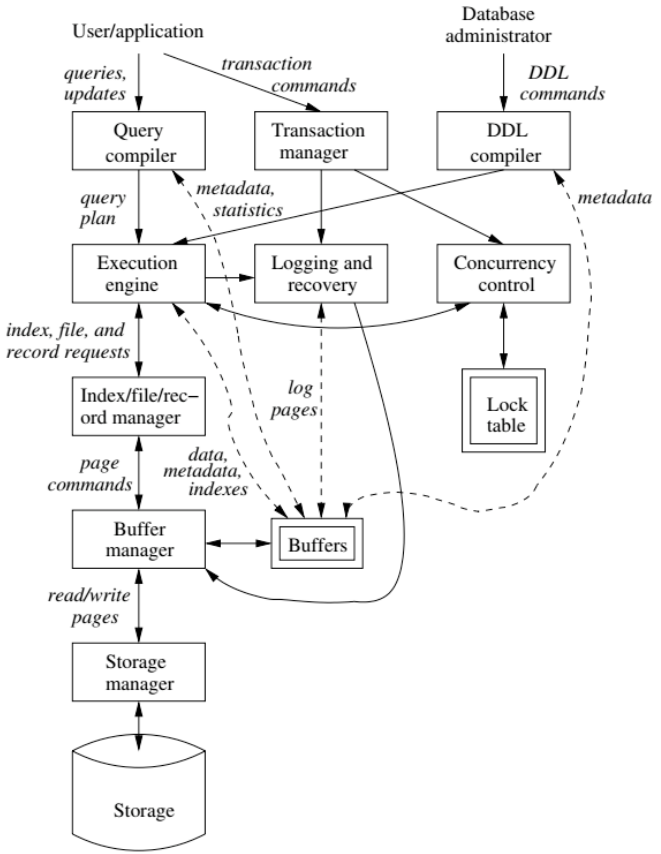
Course Update

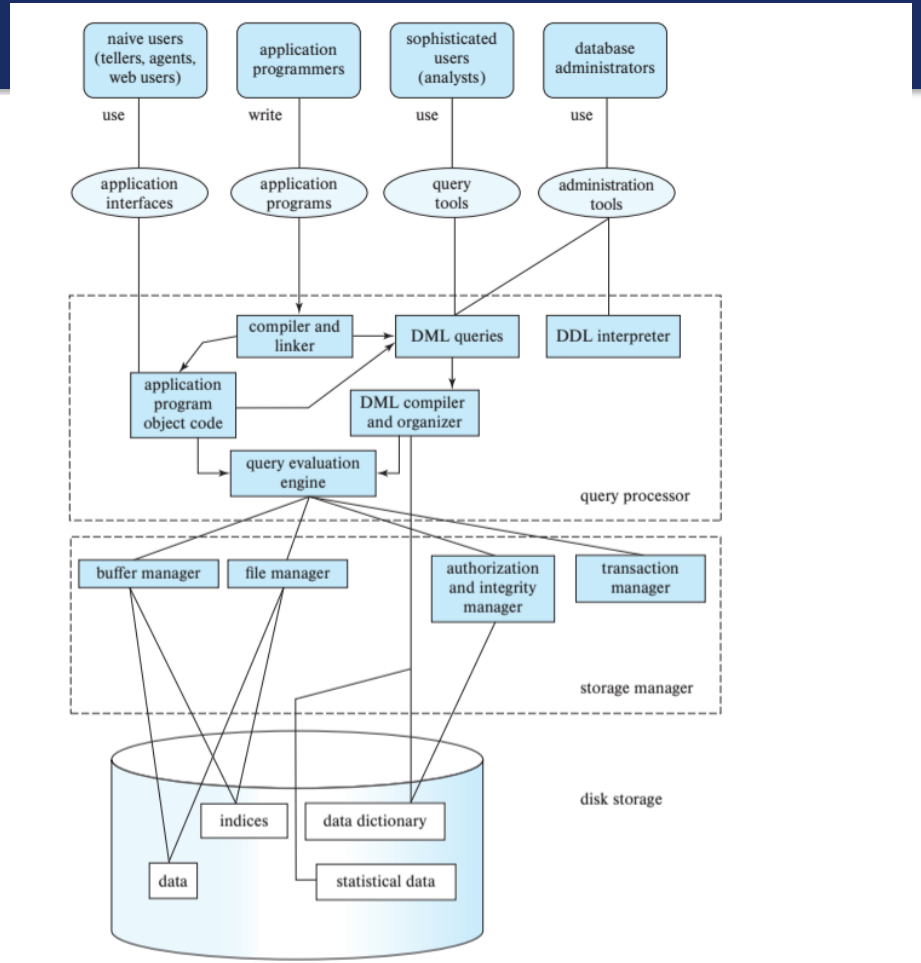
- Midterm
  - We completed midterm grading. I apologize for the delay. Some TAs had unavoidable issues that caused delays.
  - You should be able to see your results. I edited the quiz settings this morning.
  - I will hold online OHs this afternoon. I would appreciate a couple of students showing me that they see from a grading and feedback perspective.
  - I will provide a correct answer guide annotated with what I was looking for on each question.
  - I am speaking with the TAs on how to handle regrades.
- HW 4A
  - I have asked the TAs to open GradeScope for submission and provide instructions.
  - Only 34 students have watched the recitation that explains how to do the assignment.
- I am working on HW 5
  - I will incrementally publish tasks for the project starting this weekend. This will give us a chance to do the project step by step.
  - I will publish the written questions according to the schedule I provided in the previous update.

Module II – DBMS Architecture and Implementation

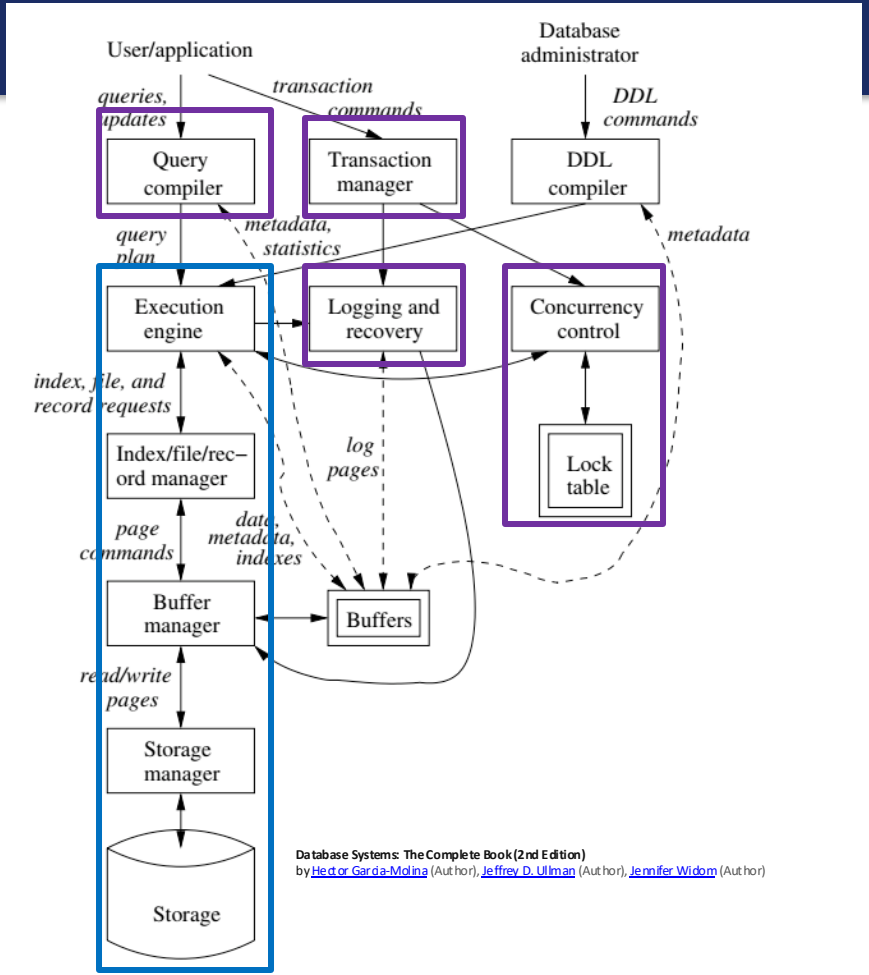
Module II – DBMS Architecture and Implementation Overview and Reminder

DBMS Arch.





Data Management

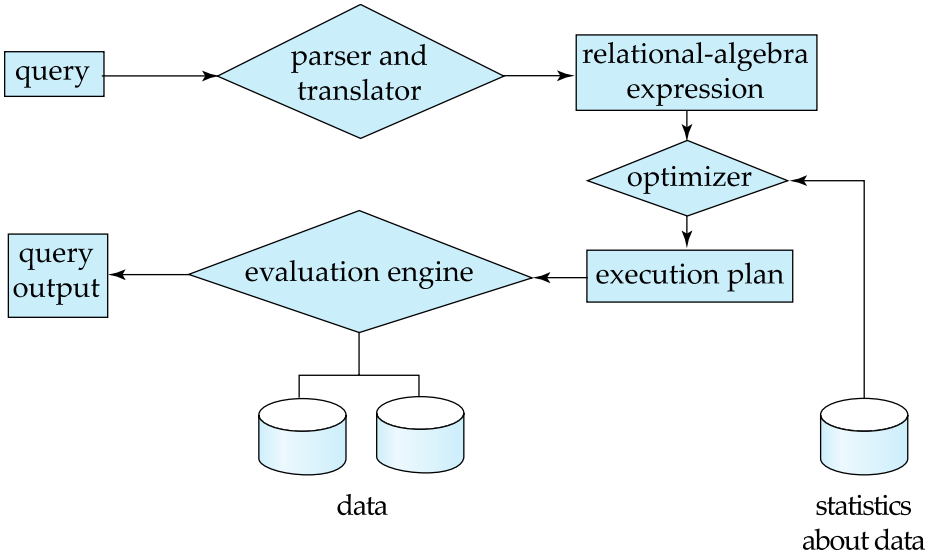


Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

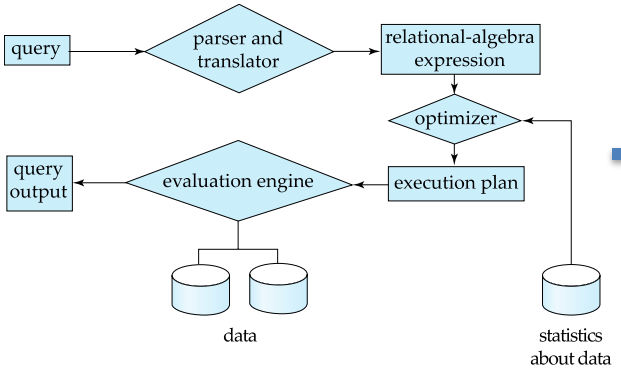
Query Processing Overview Reminder

Basic Steps in Query Processing

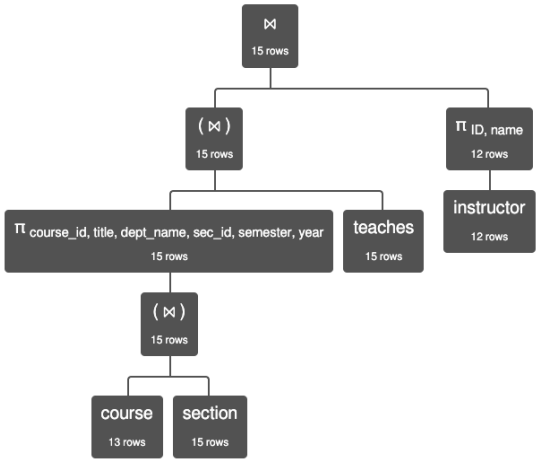
1. Parsing and translation
2. Optimization
3. Evaluation

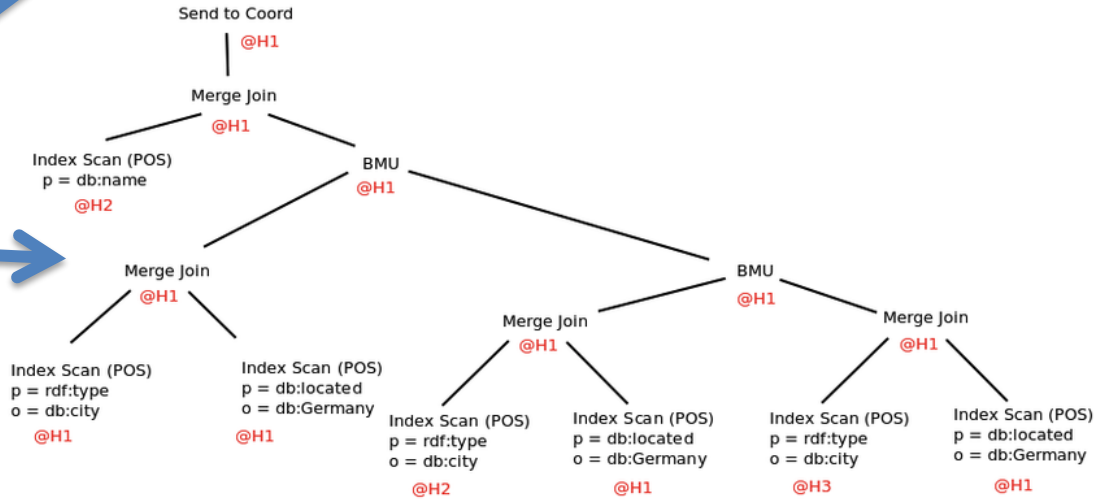


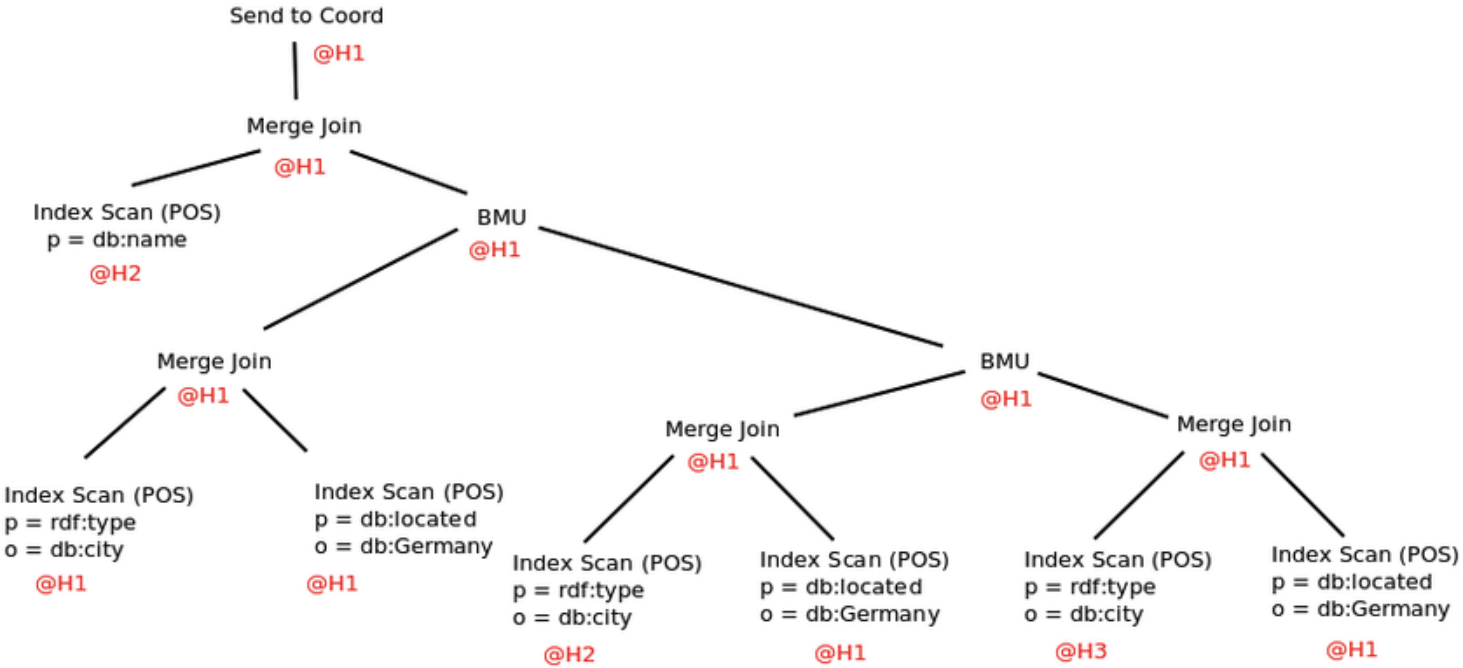
Query Processing




```
1 (
2   (π course_id, title, dept_name, sec_id, semester, year
3       (course ⋈ section)
4     )
5     ⋈ teaches
6 )
7 ⋈
8 (π ID, name (instructor))
```







Basic Steps in Query Processing: Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\prod_{salary}(instructor))$ is equivalent to $\prod_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**. E.g.,:
  - Use an index on *salary* to find instructors with salary < 75000,
  - Or perform complete relation scan and discard instructors with salary ≥ 75000

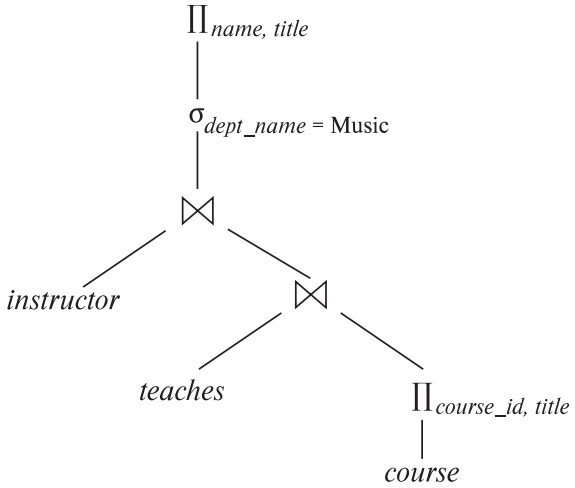
Query Optimization

Outline

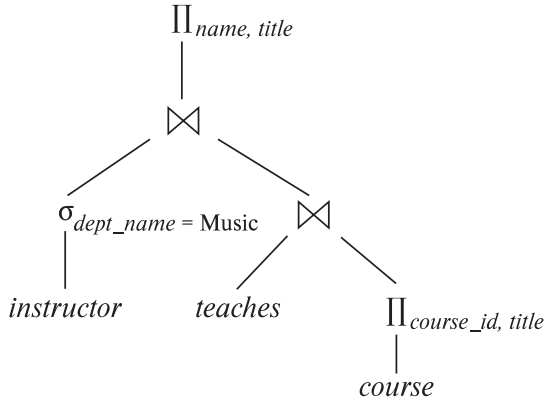
- Introduction
- Transformation of Relational Expressions
- Catalog Information for Cost Estimation
- Statistical Information for Cost Estimation
- Cost-based optimization
- Dynamic Programming for Choosing Evaluation Plans
- Materialized views

Introduction

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation



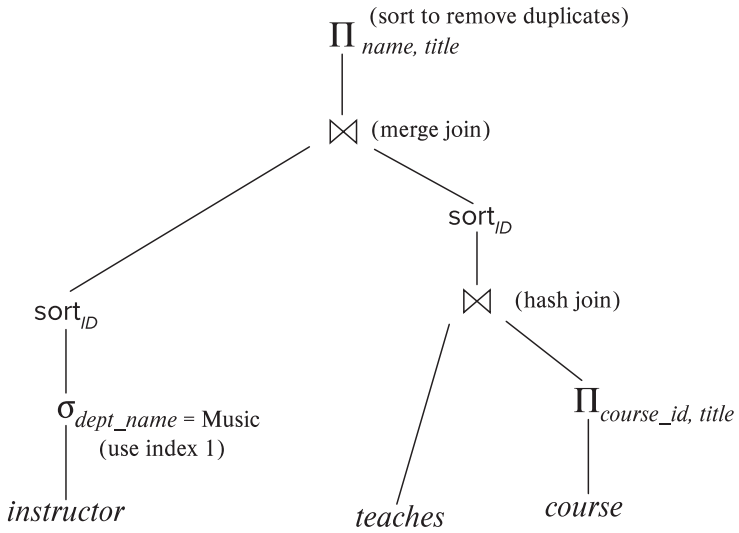
(a) Initial expression tree



(b) Transformed expression tree

Introduction (Cont.)

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



- Find out how to view query execution plans on your favorite database

Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g., seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Viewing Query Evaluation Plans

- Most database support **explain** <query>
  - Displays plan chosen by query optimizer, along with cost estimates
  - Some syntax variations between databases
    - Oracle: **explain plan for** <query> followed by **select** * **from** table (*dbms_xplan.display*)
    - SQL Server: **set showplan_text on**
- Some databases (e.g. PostgreSQL) support **explain analyse** <query>
  - Shows actual runtime statistics found by running the query, in addition to showing the plan
- Some databases (e.g. PostgreSQL) show cost as *f..l*
  - *f* is the cost of delivering first tuple and *l* is cost of delivering all results

DFF Note: Show EXPLAIN in MySQLWorkbench

Generating Equivalent Expressions

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every *legal* database instance
  - Note: order of tuples is irrelevant
  - we don't care if they generate different results on databases that violate integrity constraints
- 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 the two expressions generate the same multiset of tuples on every legal database instance.
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

   $$\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

   $$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

   $$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) \equiv \Pi_{L_1}(E)$$

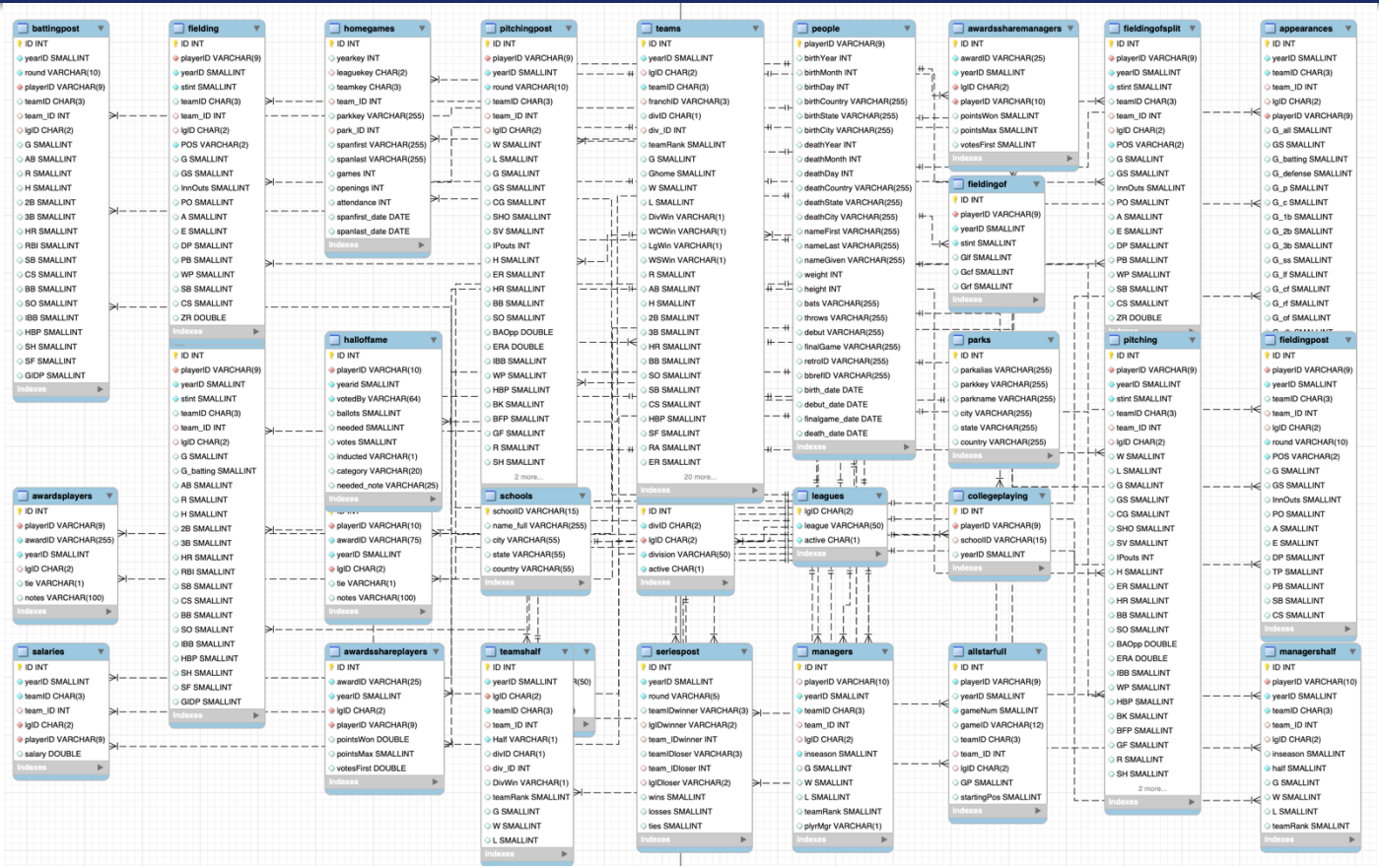
   where $L_1 \subseteq L_2 \ldots \subseteq L_n$

4. Selections can be combined with Cartesian products and theta joins.

   a. $\sigma_{\theta}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta} E_2$

   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

Lahman Baseball Database



Explanation

- $\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(E))$ –
  - What is going on here?
  - Why wouldn't we just go through the table with one scan?
- Consider Lahman's Baseball Database
  - select * from people where throws="L" and nameLast="Williams";
  - people has about 20K rows ➔ a simple scan cost is O(20K)
  - Assume there is an index on throws and an index on nameLast.
  - The engine could first use the index to get matching rows producing a smaller table, and then scan the smaller result set. But which index?
  - *Most Selective Index*

```
select count(*) as no_of_rows, count(distinct nameLast) as nameLastValue,
    count(distinct throws) as throwsValues,
    count(*)/count(distinct nameLast) as nameLastSelectivity,
    count(*)/count(distinct throws) as throwsSelectivity
from people;
```

	no_of_rows	nameLastValue	throwsValues	nameLastSelectivity	throwsSelectivity
1	19878	10135	3	1.9613	6626.0000

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

6. (a) Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

These make sense when you realize having the smaller table on the right can improve performance.

(b) Theta joins are associative in the following manner:

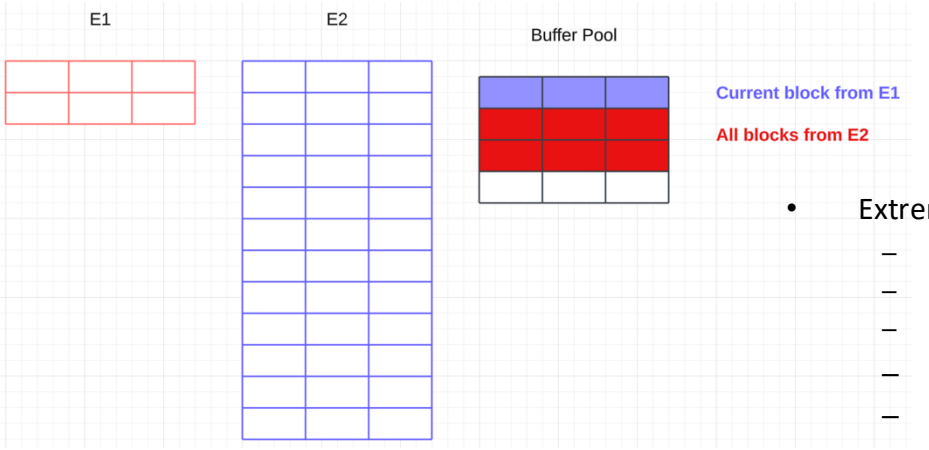
$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

This one only makes sense when you understand another rule covered in a couple of slides.

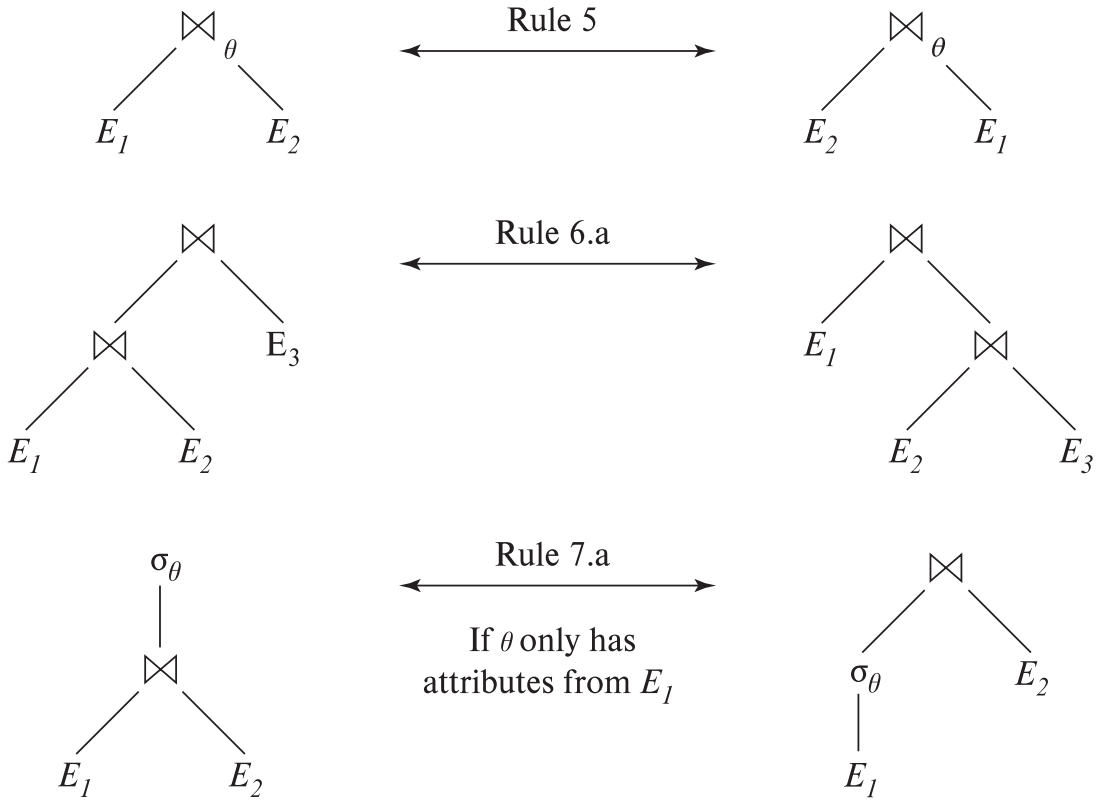
Explanation

- $E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$
- Assume there are no indexes ➔ Nested loop join ➔
  - The engine scans the left table one time.
  - The engine scans the right table once for each row in the left table.
  - If one of the tables is (much) smaller than the other, the engine prefers scanning the smaller table because it is more efficient WRT to buffer pool usage.



- Extreme example
  - $E_1 \bowtie E_2$ reads 2 blocks from E_1 and access 2*12 blocks from E_2,
  - But the engine can only have 2 or 3 in the buffer pool ➔
  - Approximately 2 + 24/3 = 10 block Ios.
  - $E_2 \bowtie E_1$ accesses 12 blocks from E_2 and 12*2 blocks from E_1
  - But all of E_1 fits in the buffer pool ➔ 14 block I/Os

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

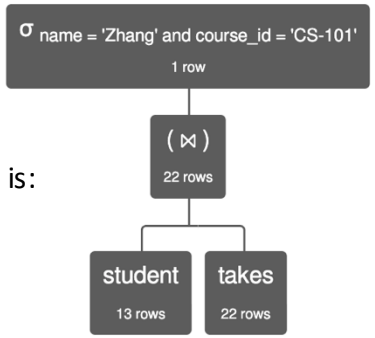
   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

Explanation

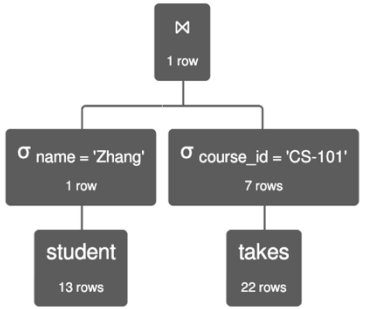
- $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$
- σ name='Zhang' ∧ course_id='CS-101' (student ⋈ takes) is equivalent to (σ name='Zhang' (student)) ⋈ (σ course_id='CS-101' (takes))

Simplistically, the cost is:
- (Join) 13 * 22 +
- (Scan) 22



σ name = 'Zhang' and course_id = 'CS-101' (student ⋈ takes)

Simplistically, the cost is:
- (Two scans) 13 + 22 +
- (Join) 1 * 7



(σ name = 'Zhang' (student)) ⋈ (σ course_id = 'CS-101' (takes))

Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

   (a) if θ involves only attributes from $L_1 \cup L_2$:

   $$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) \equiv \Pi_{L_1}(E_1) \bowtie_\theta \Pi_{L_2}(E_2)$$

   (b) In general, consider a join $E_1 \bowtie_\theta E_2$.

   - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
   - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

   $$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) \equiv \Pi_{L_1 \cup L_2}(\Pi_{L_1 \cup L_3}(E_1) \bowtie_\theta \Pi_{L_2 \cup L_4}(E_2))$$

Similar equivalences hold for outerjoin operations: $\bowtie$, $\bowtie$, and $\bowtie$

Think about this rule … …

- The project makes the buffer pool consumption smaller, and
- I can combine the rule with commutativity (from above).

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 \equiv E_2 \cup E_1$$
$$E_1 \cap E_2 \equiv E_2 \cap E_1$$

(set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

a. $\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta(E_2)$
b. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta(E_2)$
c. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta(E_2)$
d. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2$
e. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2$

preceding equivalence does not hold for $\cup$

12. The projection operation distributes over union

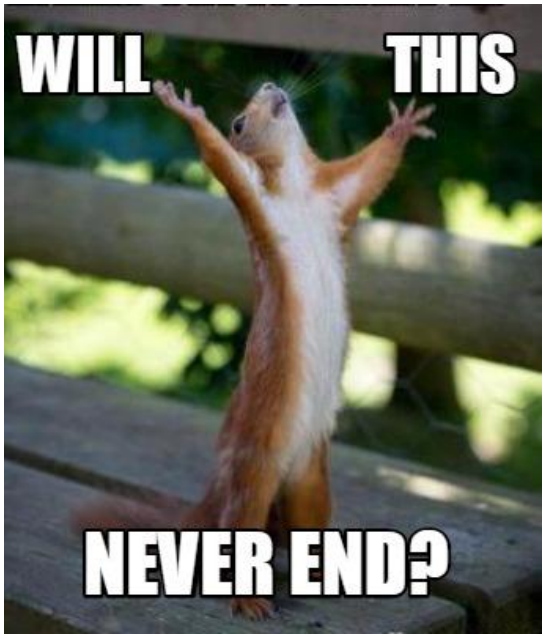
$$\Pi_L(E_1 \cup E_2) \equiv (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

I will take their word for it.

Equivalence Rules (Cont.)

13. Selection distributes over aggregation as below
    $\sigma_\theta({}_G\gamma_A(E)) \equiv {}_G\gamma_A(\sigma_\theta(E))$
    provided θ only involves attributes in G
14. a. Full outerjoin is commutative:
    $E_1 \mathbin{⟗} E_2 \equiv E_2 \mathbin{⟗} E_1$
    b. Left and right outerjoin are not commutative, but:
    $E_1 \mathbin{⟕} E_2 \equiv E_2 \mathbin{⟖} E_1$
15. Selection distributes over left and right outerjoins as below, provided θ_1 only involves attributes of E_1
    a. $\sigma_{\theta_1}(E_1 \mathbin{⟕}_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \mathbin{⟕}_\theta E_2$
    b. $\sigma_{\theta_1}(E_1 \mathbin{⟖}_\theta E_2) \equiv E_2 \mathbin{⟕}_\theta (\sigma_{\theta_1}(E_1))$
16. Outerjoins can be replaced by inner joins under some conditions
    a. $\sigma_{\theta_1}(E_1 \mathbin{⟕}_\theta E_2) \equiv \sigma_{\theta_1}(E_1 \bowtie_\theta E_2)$
    b. $\sigma_{\theta_1}(E_1 \mathbin{⟖}_\theta E_1) \equiv \sigma_{\theta_1}(E_1 \bowtie_\theta E_2)$
    provided θ_1 is null rejecting on E_2

Oh good. More rules.



Equivalence Rules (Cont.)

Note that several equivalences that hold for joins do not hold for outerjoins

- $\sigma_{year=2017}(instructor \bowtie teaches) \not\equiv \sigma_{year=2017}(instructor \bowtie teaches)$
- Outerjoins are not associative

  $(r \bowtie s) \bowtie t \quad \not\equiv \quad r \bowtie (s \bowtie t)$

  - e.g. with r(A,B) = {(1,1), s(B,C) = { (1,1)}, t(A,C) = { }

Transformation Example: Pushing Selections

- Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{'Music'}}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$
- Transformation using rule 7a.
  - $\Pi_{name,\ title}((\sigma_{dept\_name=\text{'Music'}}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course)))$
- Performing the selection as early as possible reduces the size of the relation to be joined.

Join Ordering Example

- For all relations r_1, r_2, and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

  (Join Associativity) $\bowtie$

- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

  so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

$$\Pi_{name,\ title}(\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course))))$$

- Could compute $teaches \bowtie \Pi_{course\_id,\ title}(course)$ first, and join result with

$$\sigma_{dept\_name=\text{“Music”}}(instructor)$$

but the result of the first join is likely to be a large relation.

- Only a small fraction of the university's instructors are likely to be from the Music department
  - it is better to compute

$$\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches$$

first.

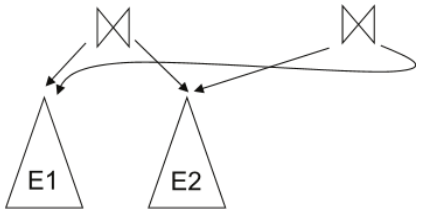
Enumeration of Equivalent Expressions

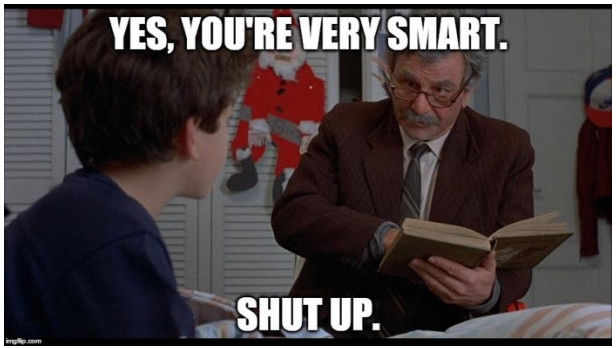
- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:
  - Repeat
    - apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
    - add newly generated expressions to the set of equivalent expressions

    Until no new equivalent expressions are generated above
- The above approach is very expensive in space and time
  - Two approaches
    - Optimized plan generation based on transformation rules
    - Special case approach for queries with only selections, projections and joins

Implementing Transformation Based Optimization

- Space requirements reduced by sharing common sub-expressions:
  - 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 using pointers
    - E.g., when applying join commutativity





  - Same sub-expression may get generated multiple times
    - Detect duplicate sub-expressions and share one copy
- Time requirements are reduced by not generating all expressions
  - Dynamic programming
    - We will study only the special case of dynamic programming for join order optimization

Cost Estimation

- Cost of each operator computer as described in Chapter 15
  - 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
- More on cost estimation later

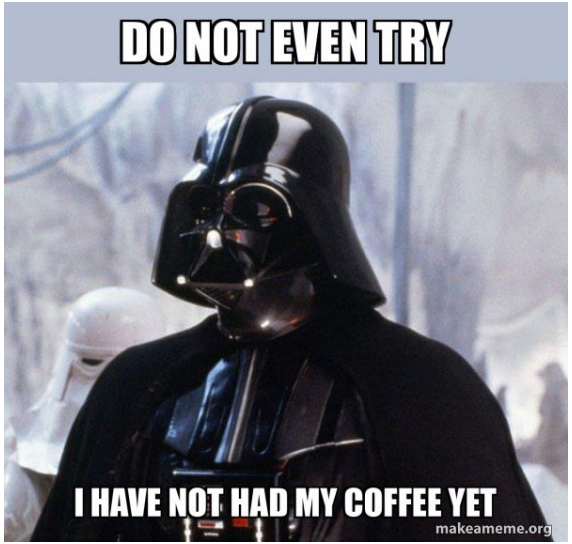
Remember algorithm selection applies to the operators in each of the possible trees.

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. Uses heuristics to choose a plan.

Join Order Optimization Algorithm

```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
            return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
       set bestplan[S].plan and bestplan[S].cost based on the best way
       of accessing S  using selections on S and indices (if any) on S else for each
    non-empty subset S1 of S such that S1 ≠ S
            P1= findbestplan(S1)
            P2= findbestplan(S - S1)
            for each algorithm A for joining results of P1 and P2
          … compute plan and cost of using A (see next page) ..
              if cost < bestplan[S].cost
                        bestplan[S].cost = cost
                        bestplan[S].plan = plan;
    return bestplan[S]
```



Statistics for Cost Estimation

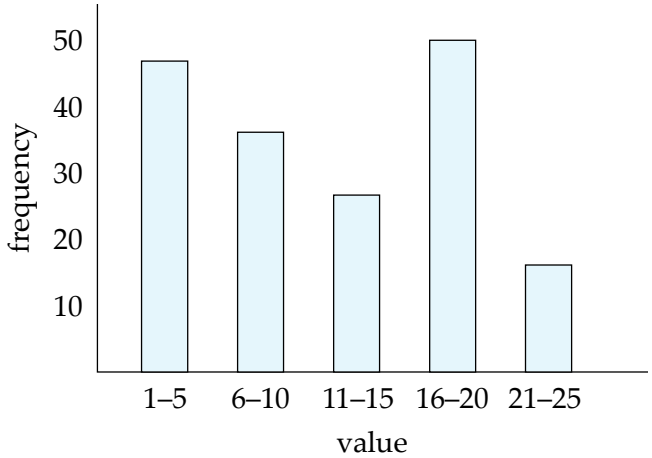
Statistical Information for Cost Estimation

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

$$b_r = \frac{n_r}{f_r}$$

Histograms

- Histogram on attribute *age* of relation *person*



- **Equi-width** histograms
- **Equi-depth** histograms break up range such that each range has (approximately) the same number of tuples
  - E.g. (4, 8, 14, 19)
- Many databases also store *n* **most-frequent values** and their counts
  - Histogram is built on remaining values only

Histograms (cont.)

- Histograms and other statistics usually computed based on a **random sample**
- Statistics may be out of date
  - Some database require a **analyze** command to be executed to update statistics
  - Others automatically recompute statistics
    - e.g., when number of tuples in a relation changes by some percentage

Selection Size Estimation

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

Size Estimation of Complex Selections

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

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

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

  n_r – *size*($\sigma_\theta(r)$)

Summary

Summary

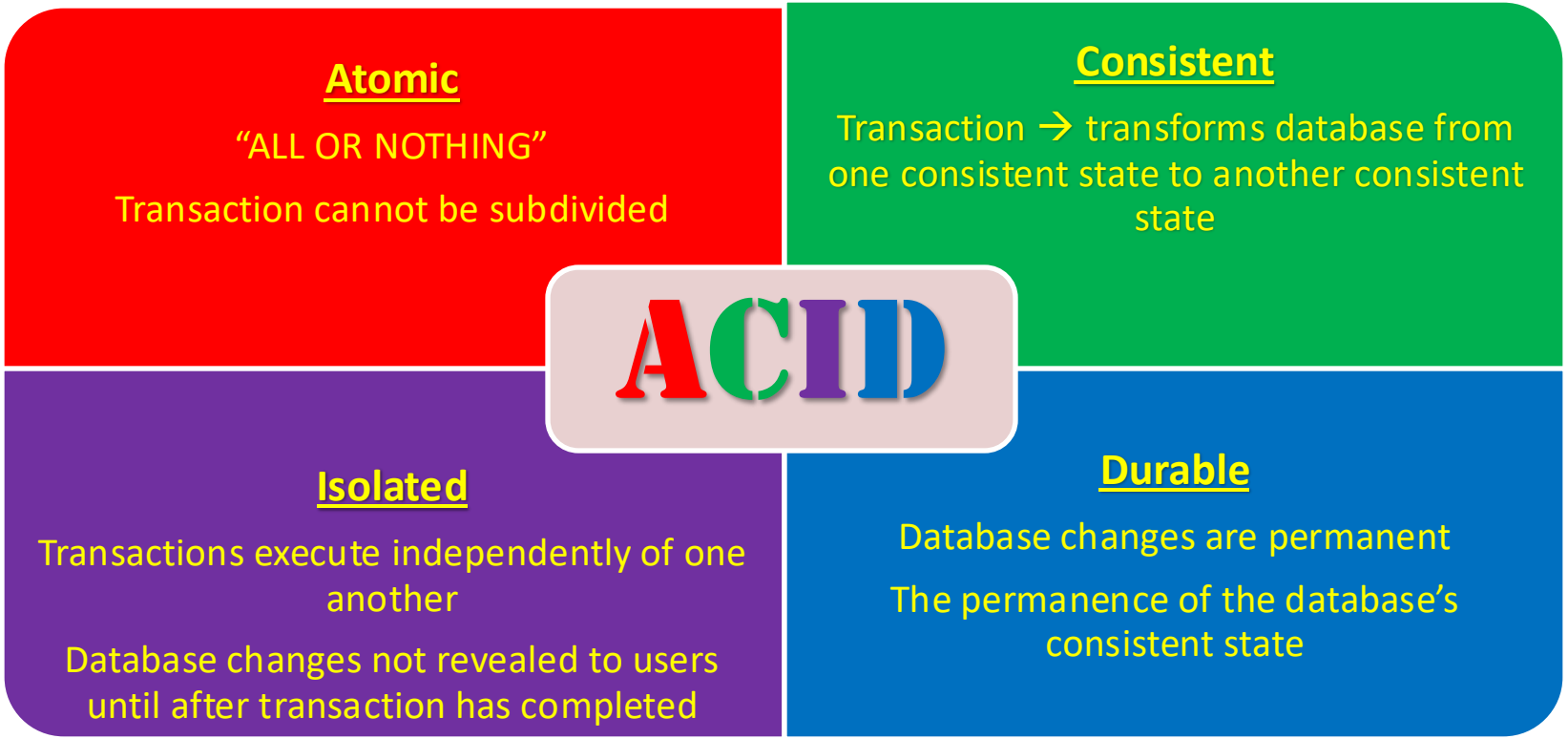
- That was painful, but … …
  - I used 33 slides but only used some of them for mocking and humor.
  - Chapter 16 has 86 slides and I skipped the complex ones.
  - And wait until you see transactions. That will make you long for the simplicity of query optimization.
- There are a handful of interrelated techniques the optimize can use:
  - Equivalent expressions.
  - Algorithm selection.
  - Parallelism for CPU and I/O.
  - Pipelining versus materialization.
- Any questions on exams will focus on your ability to think through the concepts, not memorize all of the rules and techniques.

Transactions and Recovery

Core Concepts

Core Transaction Concept is ACID Properties

http://slideplayer.com/slide/9307681



ACID Properties (httpwww.tutorialspoint.com/dbms/dbms_transaction.htm)

A *transaction* is a very small unit of a program and it may contain several low-level tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability – commonly known as ACID properties – in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** – This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** – The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** – The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** – In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer $50 from account A to account B:

BEGIN TRANSACTION {

1. **read**(*A*)
2. *A* := *A* – 50

Transfer(to_a, from_b, amount)

3. **write**(*A*)

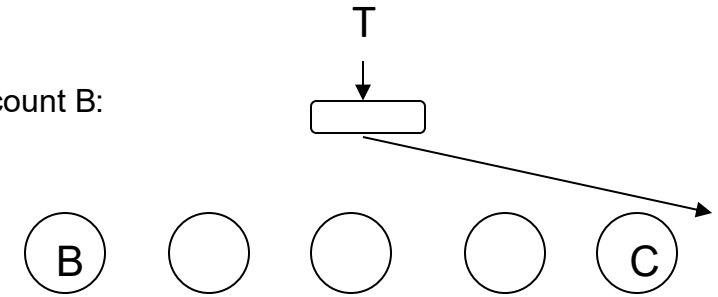
4. **read**(*B*)
5. *B* := *B* + 50

6. **write**(*B*)

COMMIT or ROLLBACK }

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

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, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **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 even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

T1, T2

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 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).

T1
T2

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	1. read(A)
	2. read(B)
	3. print("A = ", A, "B=", B, "Total=", A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B)	

- 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.

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 concurrently executing 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.

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.

Atomicity
Durability

Atomicity

A transaction is a logical unit of work that must be either entirely completed or entirely undone. (All writes happen or none of them happen)

OK. What does this mean? Consider some pseudo-code

def transfer(source_acct_id, target_acct_id, amount)

1. Check that both accounts exist.
2. IF *is_checking_account(source_acct_id)*
    1. Check that (source_acct.balance-amount) > source_account.overdraft_limit
3. ELSE
    1. Check that (source_count.balance-amount) >source_account.minimum_balance
4. Update source account
5. Update target account.
6. INSERT a record into transfer tracking table.

Atomicity

A transaction is a logical unit of work that must be either entirely completed or entirely undone. (All writes happen or none of them happen)

OK. What does this mean? Consider some pseudo-code

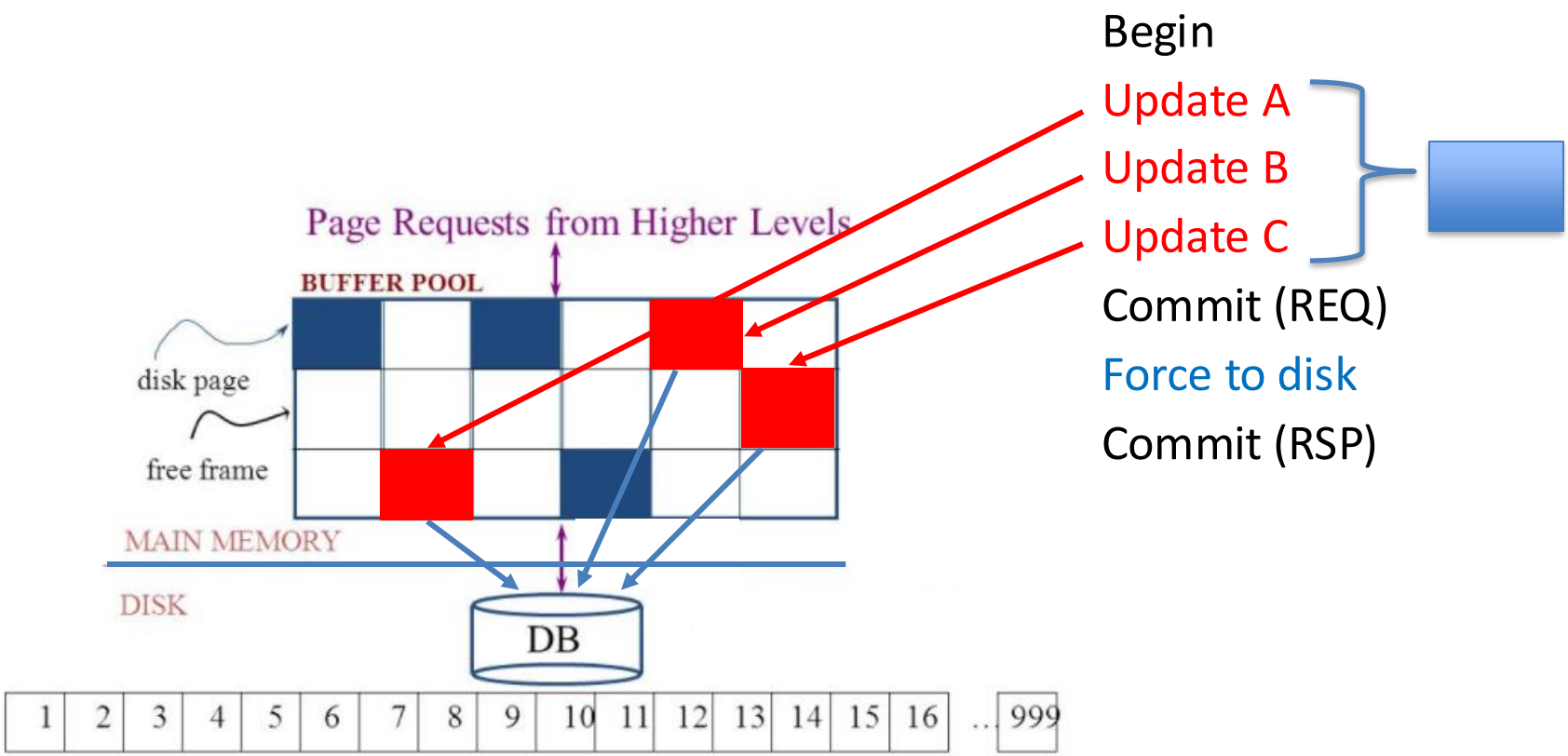
def transfer(source_acct_id, target_acct_id, amount)

1. Check that both accounts exist.
2. IF *is_checking_account(source_acct_id)*
    1. Check that (source_acct.balance-amount) > source_ac
3. ELSE
    1. Check that (source_count.balance-amount) >source_a
4. Update source account
5. Update target account.
6. INSERT a record into transfer tracking table.

Atomicity

- Transaction programs and databases are fast (milliseconds).
  What are the chances of the failure occurring in the wrong spot?
- Well, that doesn't really matter. If it happens,
  - Someone lost money and
  - There is no record off it. Someone is going to very upset.
- Even a small server can have thousands of concurrent transactions ➔
  - There will be corruptions because some transaction will be in the wrong place at the wrong time.
  - Unless we do something in the DBMS
  - Because HW and software inevitably fail
  - And sadly, SW is especially prone to failure when under load

Simplistic Approach



Simplistic Approach

There are several problems with the simplistic approach.

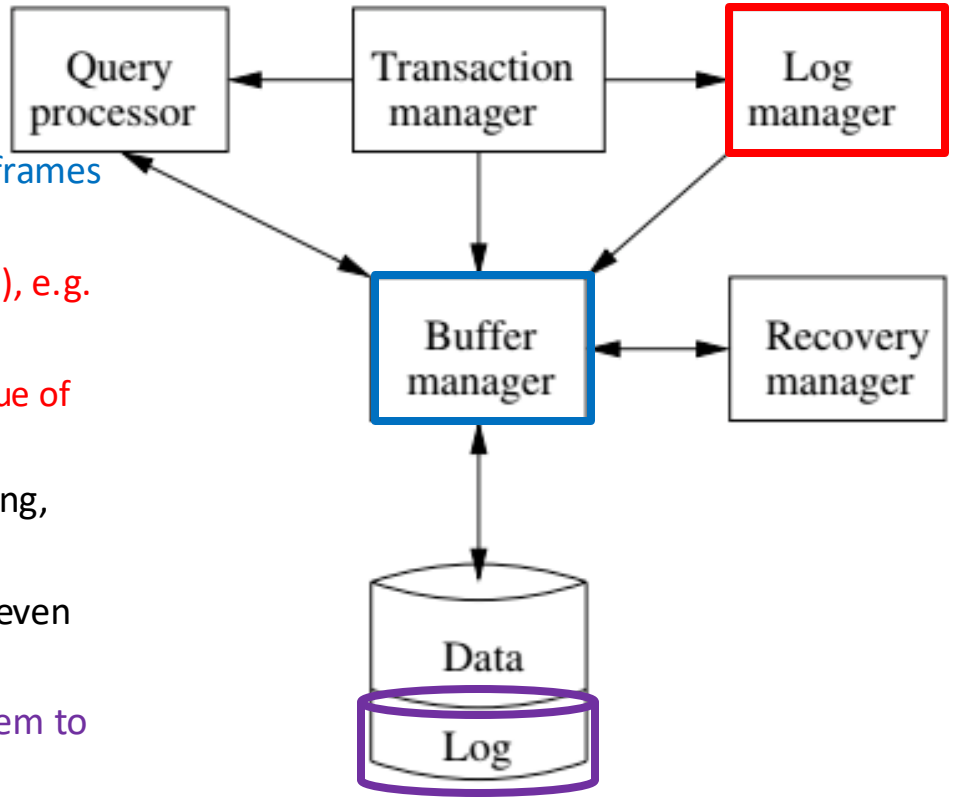
1. The approach does not solve the problem
    1. Some writes might succeed.
    2. Some might be interrupted by the failure or require retry.
2. Writes may be random and scattered. N updates might
    1. Change a few bytes in N data frames
    2. A few bytes in M index frames

    Transaction rate becomes bottlenecked by write I/O rate, even though a relatively small number of bytes change/transaction.
3. Written frames must be held in memory.
    1. Lots of transactions
    2. Randomly writing small pieces of lots of frames.
    3. Consumes lots of memory with pinned pages.
    4. Degrades the performance and optimization of the buffer.
        1. The optimal buffer replacement policy wants to hold frames that will be reused.
        2. Not frames that have been touched and never reused.

DBMS ACID Implementation

Implementation Subsystems

- *Query processor* schedules and executes queries.
- *Buffer manager* controls reading/writing blocks/frames to/from disk.
- *Log manager* journals/records events to disk (log), e.g.
  - Transaction start/commit/abort
  - Transaction update of a block and previous value of data.
- *Transaction manager* coordinates query scheduling, buffer read/write and logging to ensure ACID.
- *Recovery manager* processes log to ensure ACID even after transaction or system failures.
- *Log* is a special type of block file used by the system to optimize performance and ensure ACID.



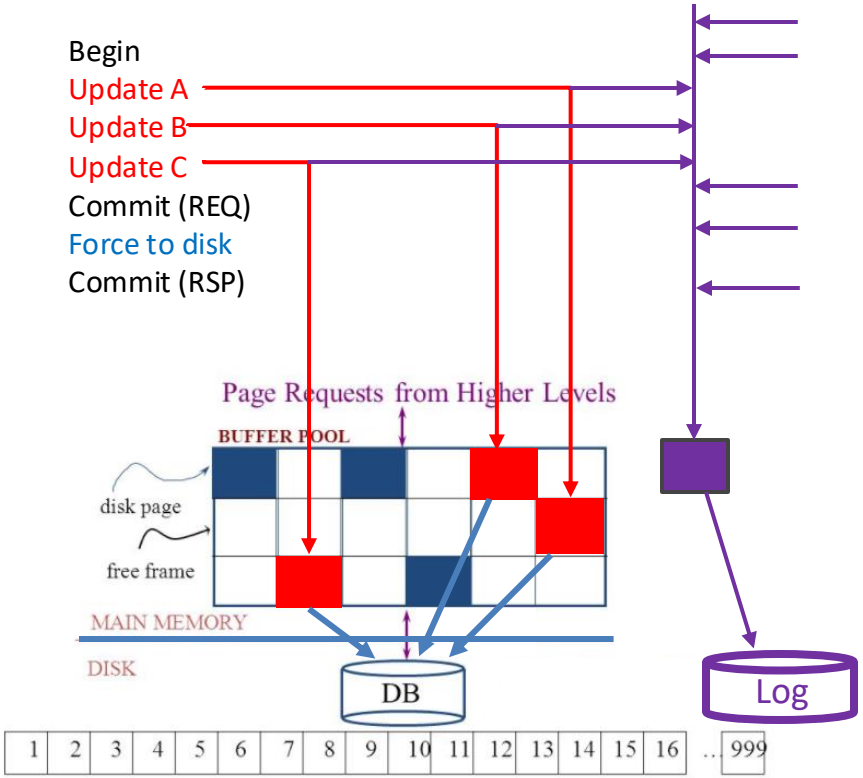
Garcia-Molina et al., p. 846

Logging

The DBMS logs every transaction event

- *Log Sequence Number* (LSN): A unique ID for a log record.
- *Prev LSN*: A link to their last log record.
- *Transaction ID number.*
- *Type*: Describes the type of database log record.
  - **Update Log Record**
    - *PageID*: A reference to the Page ID of the modified page.
    - *Length and Offset*: Length in bytes and offset of the page are usually included.
    - *Before and After Images* of records.
  - **Compensation Log Record**
  - **Commit Record**
  - **Abort Record Checkpoint Record**
  - **Completion Record** notes that all work has been done for this particular transaction.

Write Ahead Logging

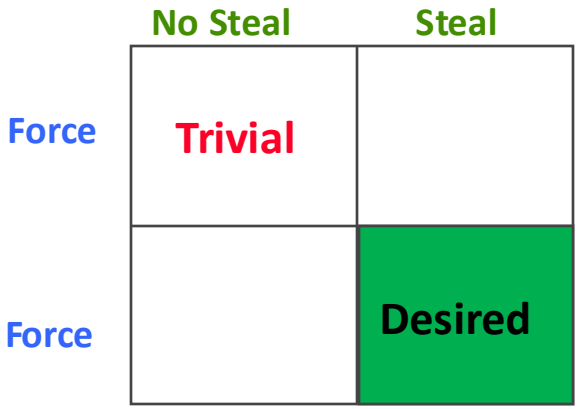


DBMS (Redo processing)

- Write log events from all transactions into a single log stream.
- Multiple events per page
- Forces (writes) log record on COMMIT/ABORT
  - Single block I/O records many updates
  - Versus multiple block I/Os, each recording a single change.
  - All of a transaction's updates recorded in one I/O versus many.
- If there is a failure
  - DBMS sequentially reads log.
  - Applies changes to modified pages that were not saved to disk.
  - Then resumes normal processing.

Write Ahead Logging

- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommitted transactions?
  - If not, poor performance/caching performance
  - If yes, how can we ensure atomicity? Uncommitted updates on disk



DBMS (Undo processing)

- Enable steal policy to improve cache performance by
  - Avoiding lots of pinned pages
  - Unlikely to be reused soon.
- Before stealing
  - Force log record to disk.
  - Update log entry has data record
    - Before image
    - After image
- If there is a failure
  - DBMS sequentially reads log.
  - Undoes changes to
    - modified pages, uncommitted pages
    - That were saved to disk.
  - Then resumes normal processing.

ARIES recovery involves three passes

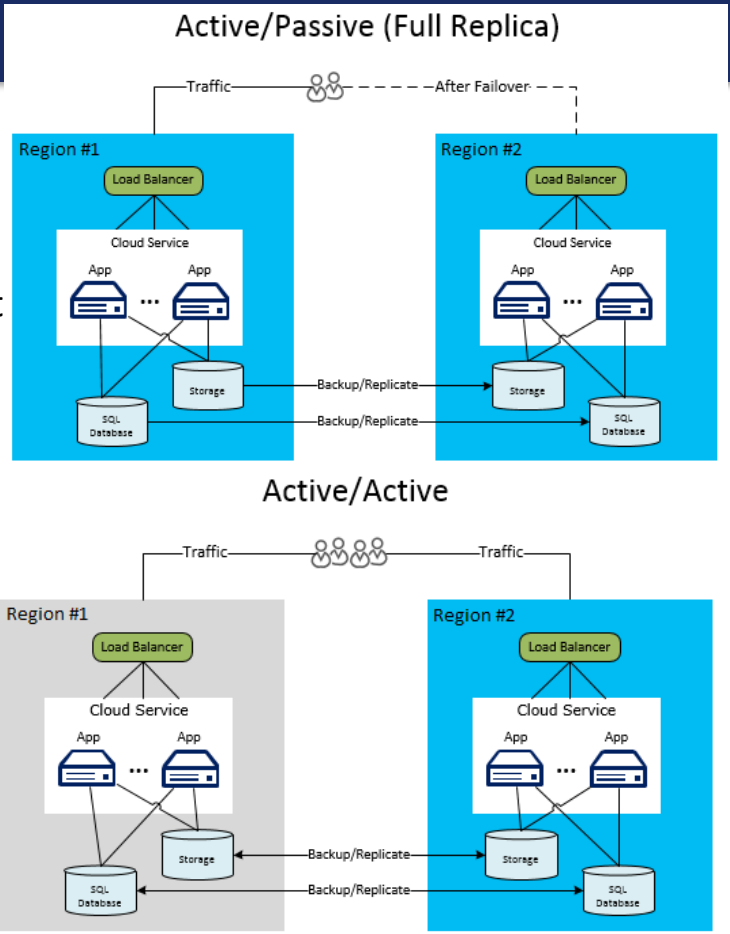
1. Analysis pass:
   - Determine which transactions to undo
   - Determine which pages were dirty (disk version not up to date) at time of
   - RedoLSN: LSN from which redo should start
2. Redo pass:
   - Repeats history, redoing all actions from RedoLSN
     (updated committed but not written changes to pages)
   - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
3. Undo pass:
   - Rolls back all incomplete transactions (with uncommitted pages written to disk).
   - Transactions whose abort was complete earlier are not undone

Durability

- Write changes to disk trivially achieves durability.
- DBMS engine uses write-ahead-logging to
  - Achieve durability
  - But with better performance through more efficient caching and I/O.
- Well, disks fail. How is that durable.
  - RAID and other solutions.
  - Disk subsystems, including entire RAID device, fail →
    - Duplex writes
    - To independent disk subsystems.
- Well, there are earthquakes, floods, etc.

Availability and Replication

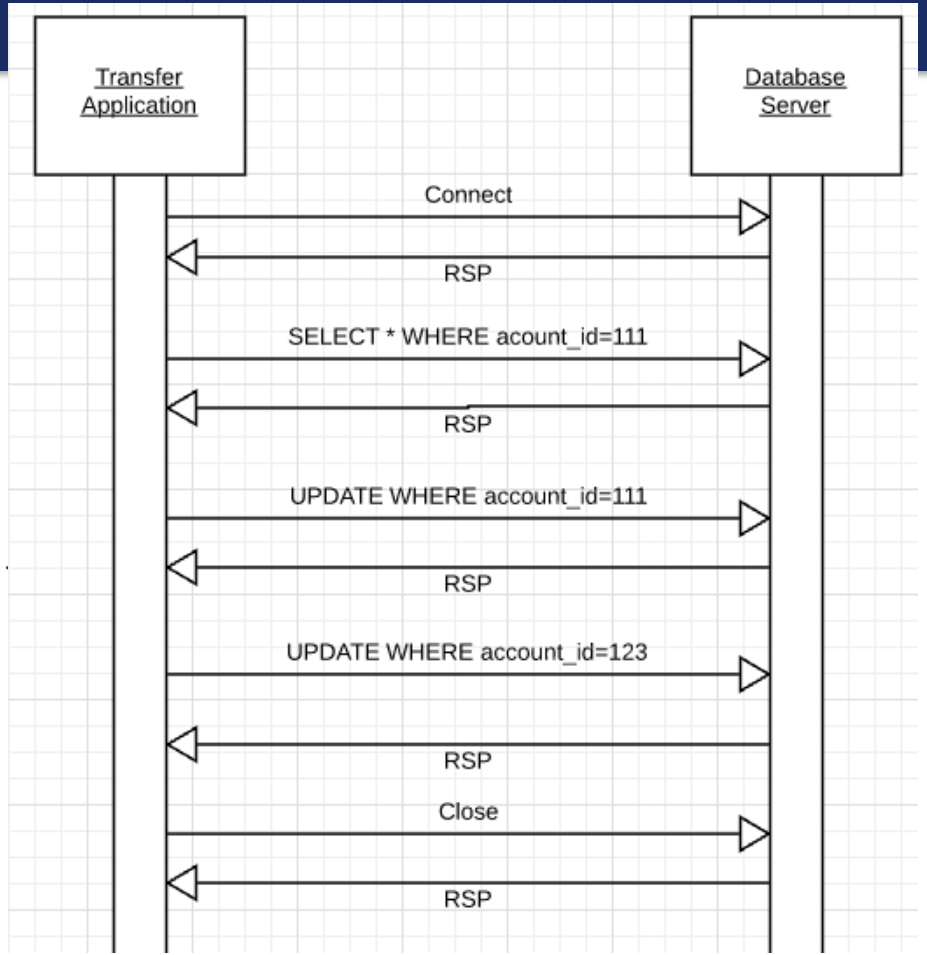
- There are two basic patterns
  - Active/Passive
    - All requests go to *master* during normal processing.
    - Updates are transactionally queued for processing at
    - Failure of *master*
      - Routes subsequent requests to *backup*.
      - Backup must process and commit updates before accepting requests.
  - Active/Active
    - Both environments process requests.
    - Some form of distributed transaction commit requir
- Multi-system communication to guarantee consis tradeoffs in CAP.
  - The system can be CAP if and only iff
  - There are never any partitions or system failures
  - Which is unrealistic in cloud/Internet systems.



Isolation

Isolation

- Transfer $50 from
  - account_id=111 to
  - account_id=123
- Requires 3 SQL statements
  - SELECT from 111 to check balance >= $50
  - UPDATE account_id=111
  - UPDATE account_id=123
- There are some interesting scenarios
  - Two different programs read the balance ($51)
  - And decide removing $50 is OK.
- DB constraints can prevent the conflict from happening, but …
  - There are more complex scenarios that constraints do not prevent.
  - Not ALL databases support constraints.
  - The "correct" execution should be that
    - One transaction responds "insufficient funds"
    - Before attempting transfer instead of after attempting.



Isolation

- Consider two **simultaneous** transfer transactions T1 and T2.
- There are two equally ***correct*** executions
  1. T1 transfers, T2 responds “insufficient funds” and does not attempt transfer
  2. T2 transfers, T1 responds “insufficient funds” and does not attempt transfer
- Each correct simultaneous execution is equivalent to a serial (sequential) execution schedule
  - (1) Execute T1, Execute T2
  - (2) Execute T2, Execute T1
- NOTE:
  - We are focusing on ***correctness*** not
  - ***Fairness:***
    - We do not care which transaction was actually submitted first.
    - And probably do not know due to networking, etc.

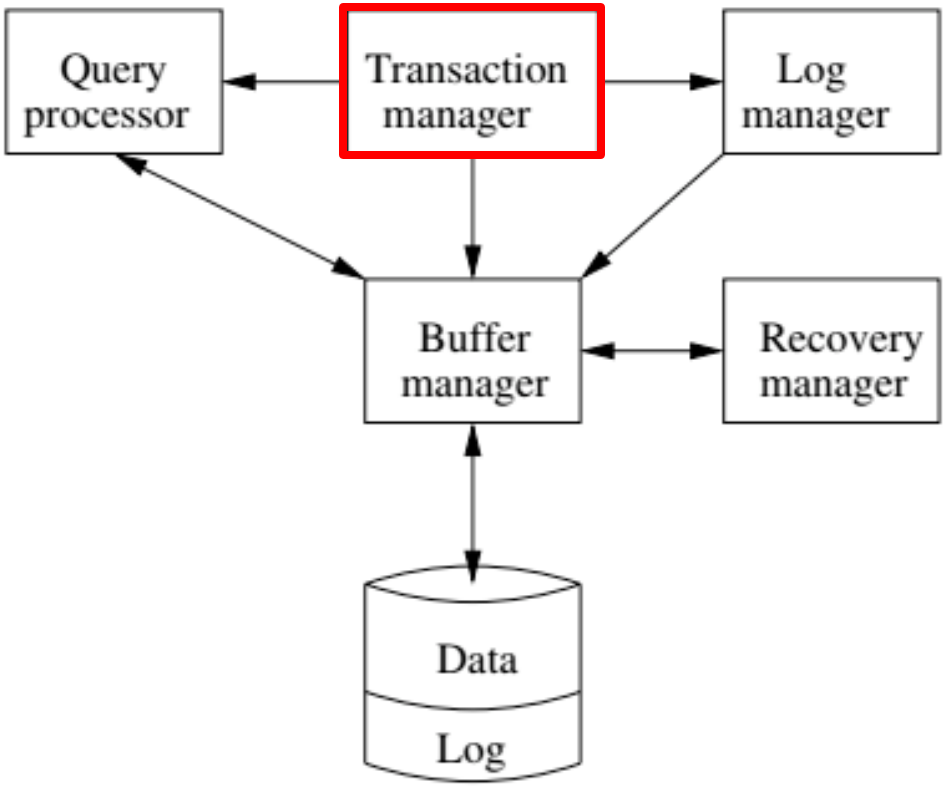
Serializability

"In concurrency control of databases,[1][2] transaction processing (transaction management), and various transactional applications (e.g., transactional memory[3] and software transactional memory), both centralized and distributed, a transaction schedule is **serializable** if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, i.e. without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems." (https://en.wikipedia.org/wiki/Serializability)

DBMS ACID Implementation

Implementation Subsystems

- *Query processor* schedules and executes queries.
- *Buffer manager* controls reading/writing blocks/frames to/from disk.
- *Log manager* journals/records events to disk (log), e.g.
  - Transaction start/commit/abort
  - Transaction update of a block and previous value of data.
- *Transaction manager* coordinates query **scheduling**, buffer read/write and logging to ensure ACID.
- *Recovery manager* processes log to ensure ACID even after transaction or system failures.
- *Log* is a special type of block file used by the system to optimize performance and ensure ACID.



Garcia-Molina et al., p. 846

Schedule

18.1.1 Schedules

A *schedule* is a sequence of the important actions taken by one or more transactions. When studying concurrency control, the important read and write actions take place in the main-memory buffers, not the disk. That is, a database element A that is brought to a buffer by some transaction T may be read or written in that buffer not only by T but by other transactions that access A.

T_1	T_2
READ(A,t)	READ(A,s)
t := t+100	s := s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

Figure 18.2: Two transactions

Garcia-Molina et al.

Serializable

- Assume there are three
  - concurrently executing transactions
  - T1, T2 and T3
- The transaction manager
  - Enables concurrent execution
  - But schedules individual operations
  - To ensure that the final DB state
  - Is *equivalent* to one of the following schedules
    - T1, T2, T3
    - T1, T3, T2
    - T2, T1, T3
    - T2, T3, T1
    - T3, T1, T2
    - T3, T2, T1

  Concurrent execution was *serializable.*

18.1.2 Serial Schedules

A schedule is *serial* if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on. No mixing of the actions is allowed.

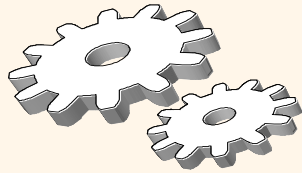
T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

Figure 18.3: Serial schedule in which T_1 precedes T_2

Serializability (en.wikipedia.org/wiki/Serializability)

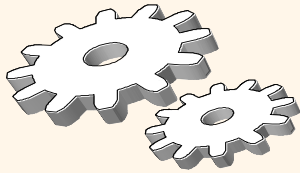
- **Serializability** is used to keep the data in the data item in a consistent state. Serializability is a property of a transaction schedule (history). It relates to the *isolation* property of a database transaction.
- **Serializability** of a schedule means equivalence (in the outcome, the database state, data values) to a *serial schedule* (i.e., sequential with no transaction overlap in time) with the same transactions. It is the major criterion for the correctness of concurrent transactions' schedule, and thus supported in all general purpose database systems.
- **The rationale behind serializability** is the following:
  - If each transaction is correct by itself, i.e., meets certain integrity conditions,
  - then a schedule that comprises any *serial* execution of these transactions is correct (its transactions still meet their conditions):
    - "Serial" means that transactions do not overlap in time and cannot interfere with each other, i.e, complete *isolation* between each other exists.
    - Any order of the transactions is legitimate, (…)
    - As a result, a schedule that comprises any execution (not necessarily serial) that is equivalent (in its outcome) to any serial execution of these transactions, is correct.

Lock-Based Concurrency Control



- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Aborting a Transaction



- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

MySQL (Locking) Isolation

13.3.6 SET TRANSACTION Syntax

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic:
    ISOLATION LEVEL level
  | READ WRITE
  | READ ONLY

level:
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
```

Scope of Transaction Characteristics

You can set transaction characteristics globally, for the current session, or for the next transaction:

- With the `GLOBAL` keyword, the statement applies globally for all subsequent sessions. Existing sessions are unaffected.
- With the `SESSION` keyword, the statement applies to all subsequent transactions performed within the current session.
- Without any `SESSION` or `GLOBAL` keyword, the statement applies to the next (not started) transaction performed within the current session. Subsequent transactions revert to using the `SESSION` isolation level.

Isolation Levels (https://en.wikipedia.org/wiki/Isolation_(database_systems)

Not all transaction use cases require 2PL and serializable execution. Databases support a a set of levels.

- **Serializable**
  - With a lock-based concurrency control DBMS implementation, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction. Also *range-locks* must be acquired when a SELECT query uses a ranged *WHERE* clause, especially to avoid the ***phantom reads*** phenomenon.
  - *The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.*
- **Repeatable reads**
  - In this isolation level, a lock-based concurrency control DBMS implementation keeps read and write locks (acquired on selected data) until the end of the transaction. However, *range-locks* are not managed, so ***phantom reads*** can occur.
  - Write skew is possible at this isolation level, a phenomenon where two writes are allowed to the same column(s) in a table by two different writers (who have previously read the columns they are updating), resulting in the column having data that is a mix of the two transactions.[3][4]
- **Read committed**
  - In this isolation level, a lock-based concurrency control DBMS implementation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed (so the ***non-repeatable reads phenomenon*** can occur in this isolation level). As in the previous level, *range-locks* are not managed.
  - Putting it in simpler words, read committed is an isolation level that guarantees that any data read is committed at the moment it is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. It makes no promise whatsoever that if the transaction re-issues the read, it will find the same data; data is free to change after it is read.
- **Read uncommitted**
  - This is the *lowest* isolation level. In this level, ***dirty reads*** are allowed, so one transaction may see *not-yet-committed* changes made by other transactions

In Databases, Cursors Define *Isolation*

- We have talked about ACID transactions

Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)
ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
DatabaseMetaData dbmd = con.getMetaData();
if (dbmd.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)
{ Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE); }
```

- Isolation
  - Determines what happens when two or more threads are manipulating the data at the same time.
  - And is defined relative to where cursors are and what they have touched.
  - Because the cursor movement determines *what you are reading or have read.*
- *But, … Cursors are client conversation state and cannot be used in REST.*

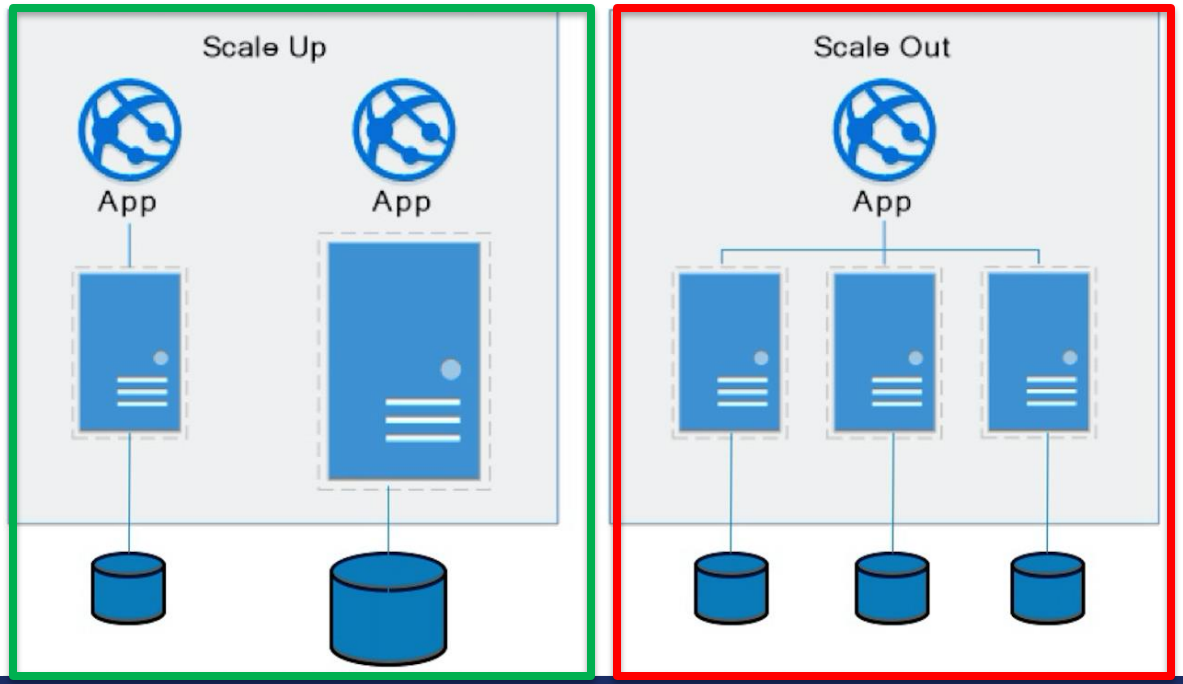
Scalability
Availability

Approaches to Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system.

Replace system with a bigger machine,
e.g. more memory, CPU, … …

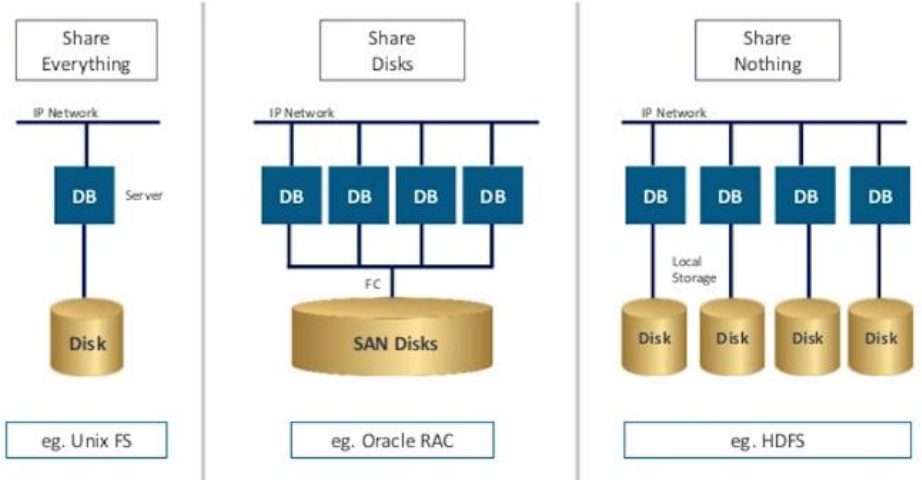
Add another system.



- Scale-up:
  - Less incremental.
  - More disruptive.
  - More expensive for extremely large systems.
  - Does not improve availability
- Scale-out:
  - Incremental cost.
  - Data replication enables availability.
  - Does not work well for functions like JOIN, referential integrity, … …

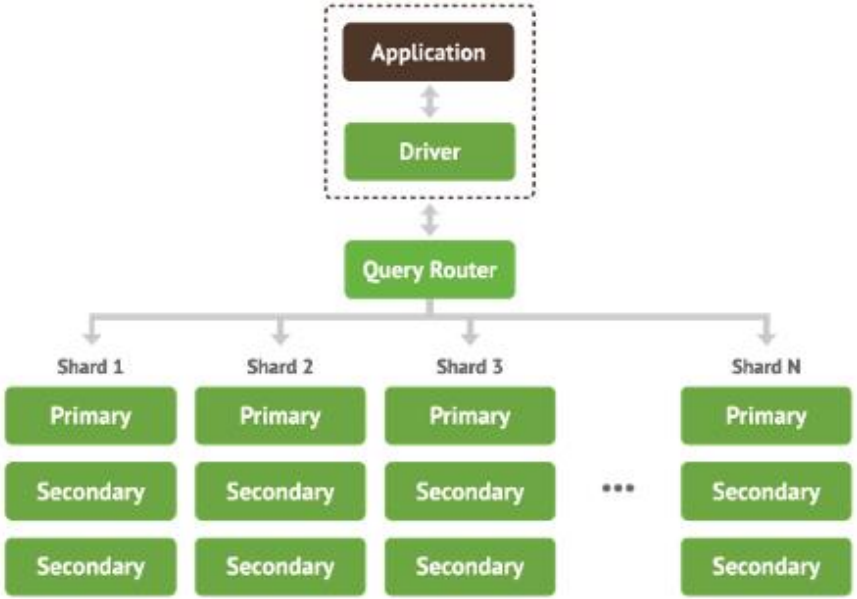
Disk Architecture for Scale-Out

- Share disks:
  - Is basically scale-up for data/disks. You can use NAS, SAN and RAID.
  - Isolation/Integrity requires distributed locking to control access from multiple database servers.
- Share nothing:
  - Is basically scale-out for disks.
  - Data is partitioned into *shards* based on a function f() applied to a key.
  - Can improve availability, at the code consistency, with data replication.
  - There is a router that sends requests to the proper shard based on the function.

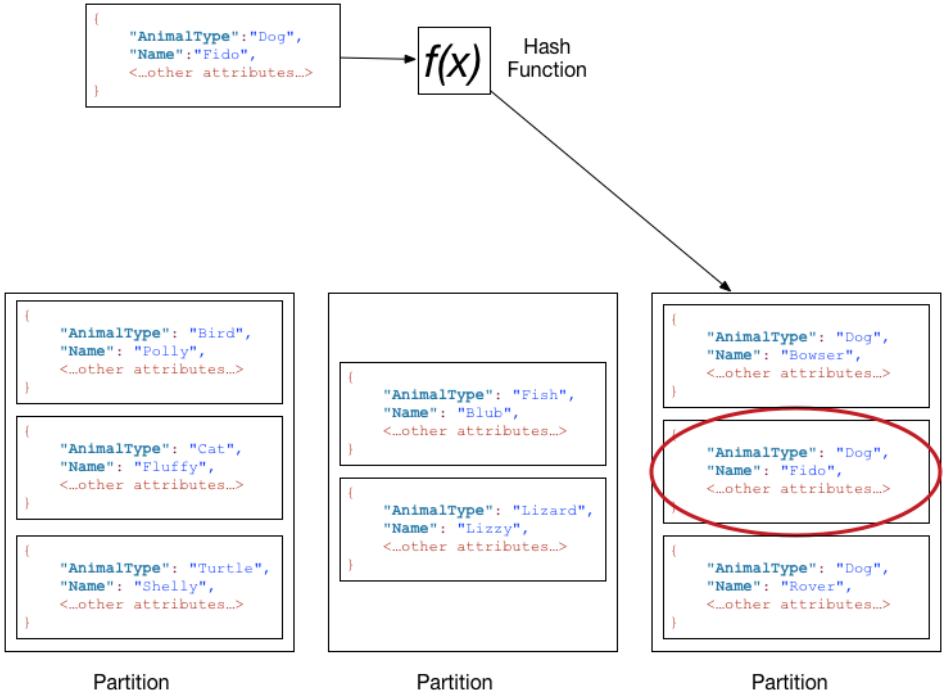


Shared Nothing, Scale-Out

MongoDB Sharding



DynamoDB Partitioning



Web Applications
REST

Full Stack Application

Full Stack Developer Meaning & Definition

In technology development, full stack refers to an entire computer system or application from the front end to the back end and the code that connects the two. The back end of a computer system encompasses "behind-the-scenes" technologies such as the database and operating system. The front end is the user interface (UI). This end-to-end system requires many ancillary technologies such as the network, hardware, load balancers, and firewalls.

FULL STACK WEB DEVELOPERS

Full stack is most commonly used when referring to web developers. A full stack web developer works with both the front and back end of a website or application. They are proficient in both front-end and back-end languages and frameworks, as well as server, network, and hosting environments.

Full-stack developers need to be proficient in languages used for front-end development such as HTML, CSS, JavaScript, and third-party libraries and extensions for Web development such as JQuery, SASS, and REACT. Mastery of these front-end programming languages will need to be combined with knowledge of UI design as well as customer experience design for creating optimal front-facing websites and applications.

https://www.webopedia.com/definitions/full-stack/

Full Stack Web Developer

A full stack web developer is a person who can develop both **client** and **server** software.

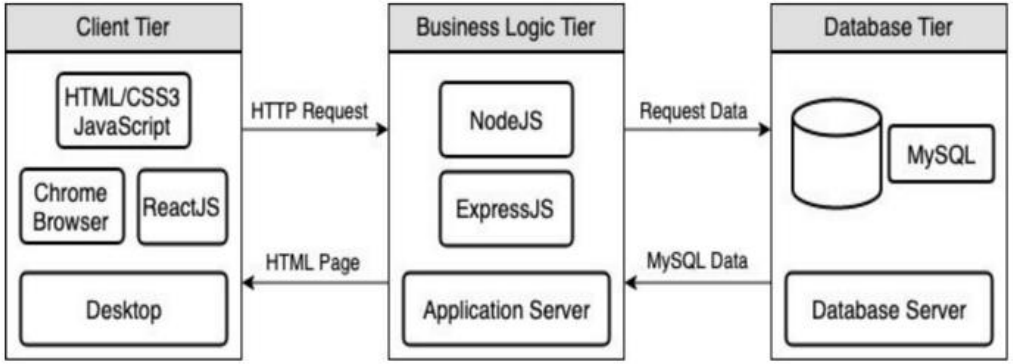
In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (like using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (like using PHP, ASP, Python, or Node)
- Program a **database** (like using SQL, SQLite, or MongoDB)

https://www.w3schools.com/whatis/whatis_fullstack.asp

- There are courses that cover topics:
  - COMS W4153: Advanced Software Engineering
  - COMS W4111: Introduction to Databases
  - COMS W4170 - User Interface Design
- This course will focus on cloud realization, microservices and application patterns, … …
- Also, I am not great at UIs … ... We will not emphasize or require a lot of UI work.

Full Stack Web Application



M = Mongo
E = Express
R = React
N = Node

I start with FastAPI and MySQL, but all the concepts are the same.

https://levelup.gitconnected.com/a-complete-guide-build-a-scalable-3-tier-architecture-with-mern-stack-es6-ca129d7df805

- My preferences are to replace React with Angular, and Node with Flask.
- There are three projects to design, develop, test, deploy, … ...
  1. Browser UI application.
  2. Microservice.
  3. Database.
- We will initial have two deployments: local machine, virtual machine. We will ignore the database for step 1.

Some Terms

- A web application server or web application framework: "A web framework (WF) or web application framework (WAF) is a software framework that is designed to support the development of web applications including web services, web resources, and web APIs. Web frameworks provide a standard way to build and deploy web applications on the World Wide Web. Web frameworks aim to automate the overhead associated with common activities performed in web development. For example, many web frameworks provide libraries for database access, templating frameworks, and session management, and they often promote code reuse." (https://en.wikipedia.org/wiki/Web_framework)

- REST: "REST (Representational State Transfer) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of a distributed, Internet-scale hypermedia system, such as the Web, should behave. The REST architectural style emphasises uniform interfaces, independent deployment of components, the scalability of interactions between them, and creating a layered architecture to promote caching to reduce user-perceived latency, enforce security, and encapsulate legacy systems.[1]

  REST has been employed throughout the software industry to create stateless, reliable web-based applications." (https://en.wikipedia.org/wiki/REST)

Some Terms

- OpenAPI: “The OpenAPI Specification, previously known as the Swagger Specification, is a specification for a machine-readable interface definition language for describing, producing, consuming and visualizing web services.” (https://en.wikipedia.org/wiki/OpenAPI_Specification)

- Model: “A model represents an entity of our application domain with an associated type.” (https://medium.com/@nicola88/your-first-openapi-document-part-ii-data-model-52ee1d6503e0)

- Routers: “What fastapi docs says about routers: If you are building an application or a web API, it’s rarely the case that you can put everything on a single file. FastAPI provides a convenience tool to structure your application while keeping all the flexibility.” (https://medium.com/@rushikeshnaik779/routers-in-fastapi-tutorial-2-adf3e505fdca)

- Summary:
  - These are general concepts, and we will go into more detail in the semester.
  - FastAPI is a specific technology for Python.
  - There are many other frameworks applicable to Python, NodeJS/TypeScript, Go, C#, Java, … …
  - They all surface similar concepts with slightly different names.

REST (https://restfulapi.net/)

What is REST

- REST is acronym for **RE**presentational **S**tate **T**ransfer. It is architectural style for **distributed hypermedia systems** and was first presented by Roy Fielding in 2000 in his famous dissertation.
- Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as **RESTful**. These principles are listed below.

Guiding Principles of REST

- **Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
- **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
- **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
- **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.
- **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

Resources

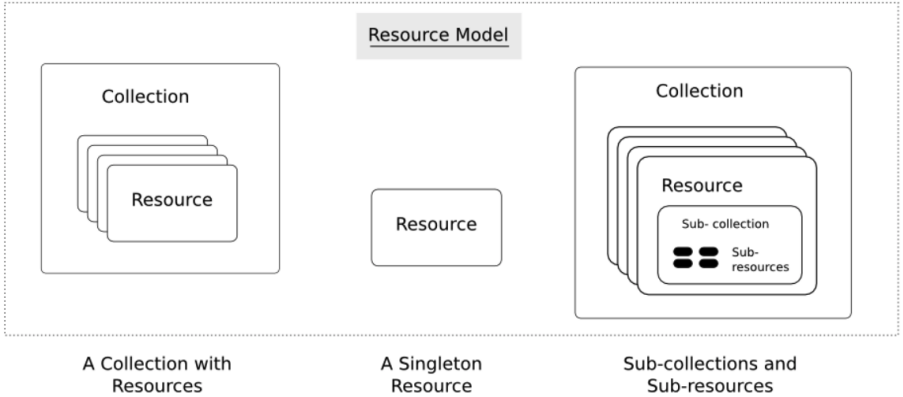
Resources are an abstraction. The application maps to create things and actions.

"A resource-oriented API is generally modeled as a resource hierarchy, where each node is either a *simple resource* or a *collection resource*. For convenience, they are often called a resource and a collection, respectively.

- A collection contains a list of resources of **the same type**. For example, a user has a collection of contacts.
- A resource has some state and zero or more sub-resources. Each sub-resource can be either a simple resource or a collection resource.

For example, Gmail API has a collection of users, each user has a collection of messages, a collection of threads, a collection of labels, a profile resource, and several setting resources.

While there is some conceptual alignment between storage systems and REST APIs, a service with a resource-oriented API is not necessarily a database, and has enormous flexibility in how it interprets resources and methods. For example, creating a calendar event (resource) may create additional events for attendees, send email invitations to attendees, reserve conference rooms, and update video conference schedules. (Emphasis added) (https://cloud.google.com/apis/design/resources#resources)



https://restful-api-design.readthedocs.io/en/latest/resources.html

REST – Resource Oriented

- When writing applications, we are used to writing functions or methods:
  - openAccount(last_name, first_name, tax_payer_id)
  - account.deposit(deposit_amount)
  - account.close()

  We can create and implement whatever functions we need.
- REST only allows four methods:
  - POST: Create a resource
  - GET: Retrieve a resource
  - PUT: Update a resource
  - DELETE: Delete a resource

  That's it. That's all you get.

"The key characteristic of a resource-oriented API is that it emphasizes resources (data model) over the methods performed on the resources (functionality). A typical resource-oriented API exposes a large number of resources with a small number of methods." (https://cloud.google.com/apis/design/resources)

- A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

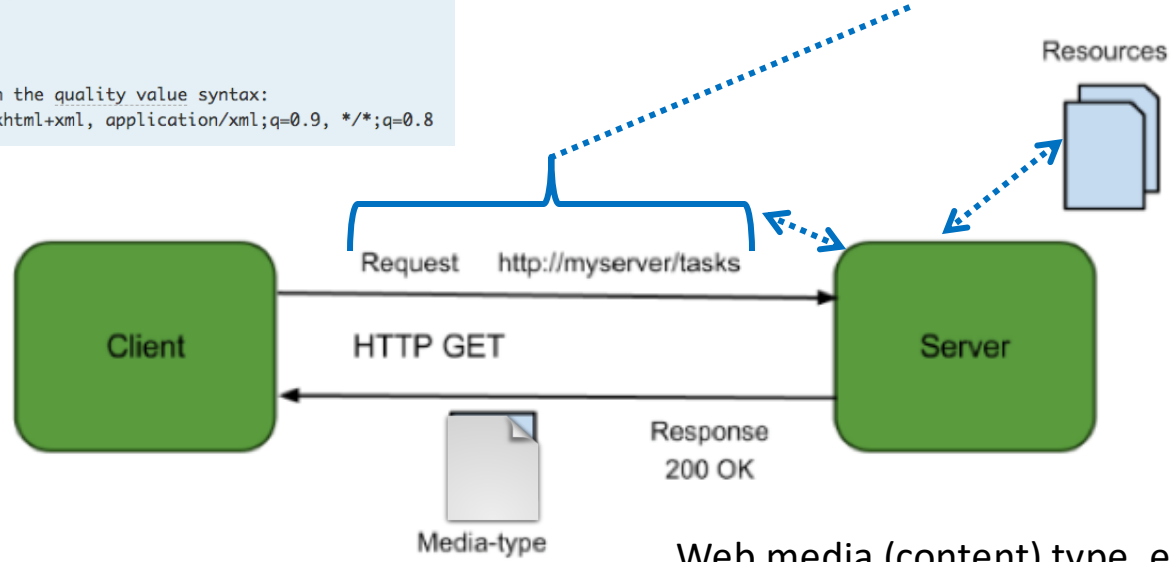
Resources, URLs, Content Types

- Relative URL identifies “resource” on the server.
- Server implementation maps abstract resource to tangible “thing,” file, DB row, … and any application logic.

Accept type in headers.

```
Accept: <MIME_type>/<MIME_subtype>
Accept: <MIME_type>/*
Accept: */*

// Multiple types, weighted with the quality value syntax:
Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8
```



Client may be

Browser

Mobile device

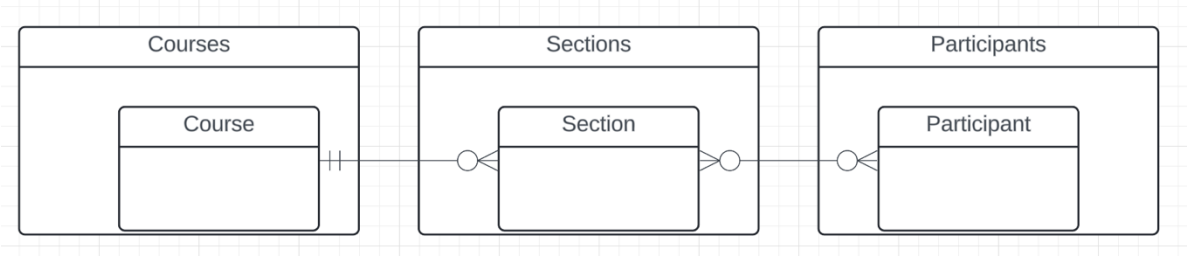
Other REST Service

… …

Web media (content) type, e.g.

- text/html
- application/json

Resources and APIs



- Base resources, paths and methods:
  - /courses: GET~~, POST~~
  - /courses/<id>: GET, ~~PUT, DELETE~~
  - /sections: GET, ~~POST~~
  - /sections/<id>: GET, ~~PUT, DELETE~~
  - /participants: GET, ~~POST~~
  - /participants/<id>: GET, ~~PUT, DELETE~~
- There are relative, navigation paths:
  - /courses/<id>/sections
  - /participants/<id>/sections
  - etc.
- There are two approaches to defining API
  - Start with OpenAPI document and produce an implementation template.
  - Start with annotated code and generate API document.
- In either approach, I start with *models.*
- Also,
  - I lack the security permission to update CourseWorks.
  - I can choose to not surface the methods or raise and exception.
- GET on resources that are collections may also have query parameters.

Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains.***
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...
  - Host/database/table/pk

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)

What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – Provides a read only access to a resource.
- **POST** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **PUT** – Used to update a existing resource or create a new resource.

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)

Introduction to RESTFul web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

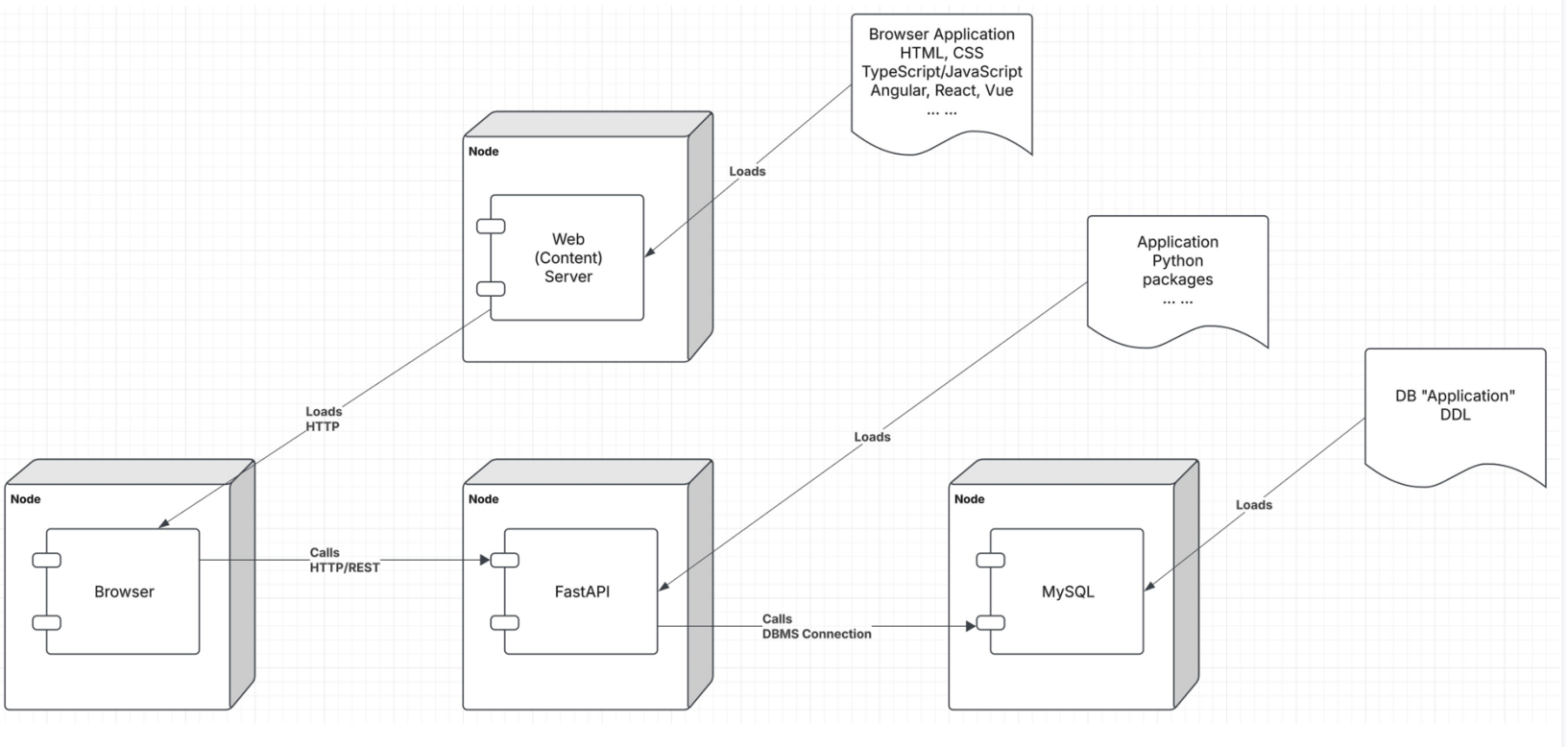
Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

Creating RESTFul Webservice

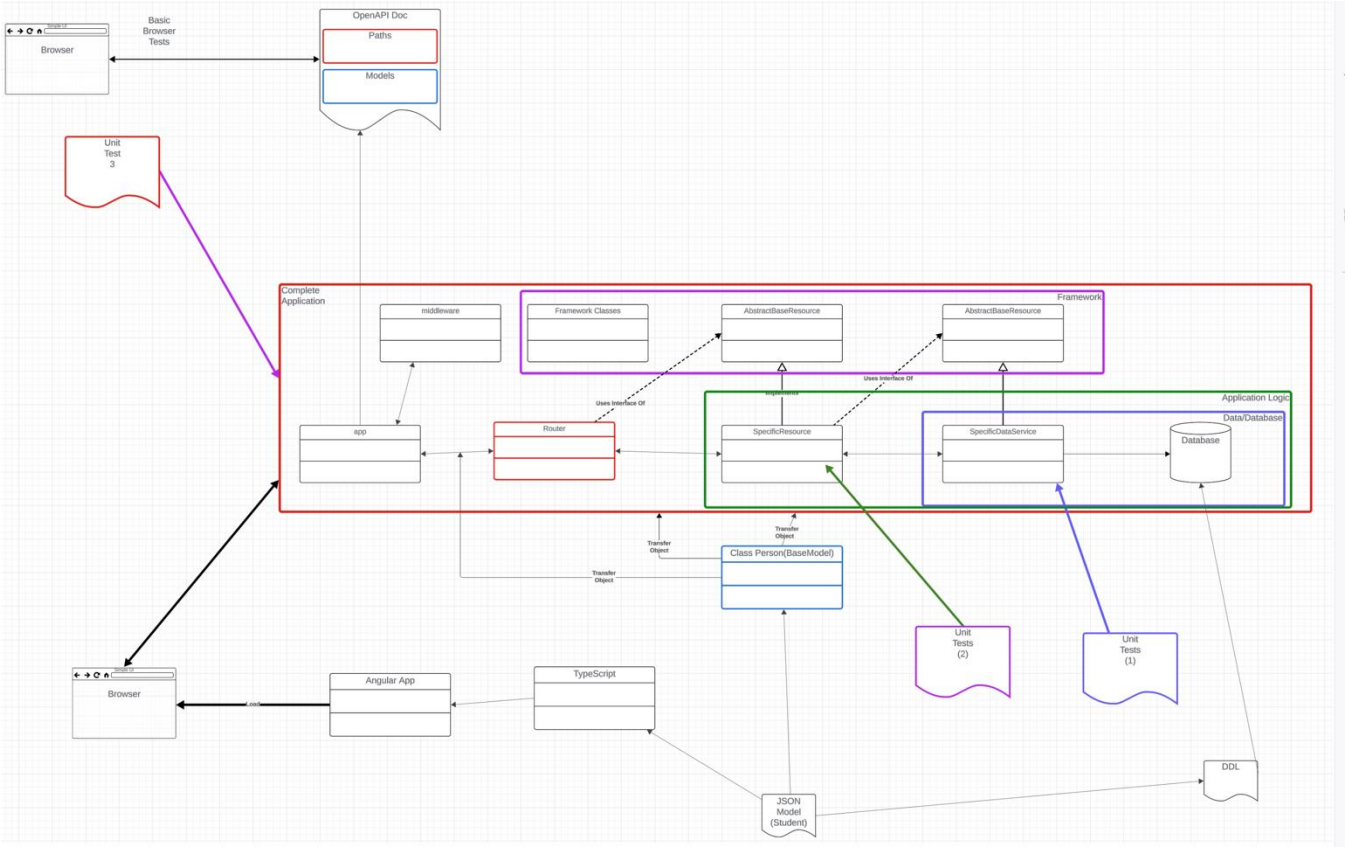
In next chapters, we'll create a webservice say user management with following functionalities –

Sr.No.	URI	HTTP Method	POST body	Result
1	/UserService/users	GET	empty	Show list of all the users.
2	/UserService/addUser	POST	JSON String	Add details of new user.
3	/UserService/getUser/:id	GET	empty	Show details of a user.

Let’s Look at the Programming Project



Let’s Look at the Programming Project



Let’s Look at the Programming Project