1. Introduction

- References to SQL and relational databases appeared in several places in previous modules. Let's take a closer look now.
- The relationship between MapReduce and relational database management systems (DBMS) has been the subject of heated arguments ever since MapReduce was proposed. [Link: Read what two DBMS luminaries thought about MapReduce and how readers reacted.]
- In the end, despite all differences, there are many commonalities between the two contenders. In fact, the relationship between DBMS and MapReduce is an active research area in the database community to find ways for combining the best of both worlds.

1.1 DBMS Overview

- Relational databases dominate the database market. Hence we will sometimes simply say “database” or “DBMS”, automatically implying that they are based on relational technology.
- Relational databases have been around since 1970s. Parallel and distributed database systems were actively researched in the 1980s and are making a strong comeback. DBMSs have been highly successful and ubiquitous, including for query-intensive massive data warehousing applications.
- The main reason for their success lies in the use of a declarative approach to programming: The user specifies constraints describing WHAT result she wants, not HOW to compute it. Given the former, an optimizer automatically finds an efficient plan for the latter.
- The declarative approach also enables strong data independence: The user writes the query against the logical schema, not the physical layer. Hence queries do not need to be modified if data is partitioned differently in the physical layer, e.g., when a large file is split into multiple chunks. In general a database user does not need to be aware of the physical data layout at all.
- Like MapReduce, DBMSs are designed for managing and analyzing big data.

1.2 Relational Data Model

- Databases rely on a simple data representation, called a relation. It is essentially a "flat" table with a schema. A schema defines the attribute name and type for each column of the table. Table rows are called tuples. Each tuple has the same type according to the relation's schema. There is no nesting and there are no pointers. Complex objects often need to be represented with multiple relations.
- Consider an example database consisting of three relations:
  - Students(SID, Name, Age, GPA) stores student information.
  - Books(BookID, Topic, Title) stores information about books in the library.
  - Reservations(SID, BookID, Date) records which student reserved which book.
- Notice how the SID and BookID attributes relate the corresponding student and book tuples with each of the reservations tuples. Even though the relational model does not support actual pointers, i.e., memory references for directly retrieving the data, attributes like SID and BookID practically fill the role of pointers.

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8

SID	BookID	Date
2	B10	01/17/12
3	B11	01/18/12

BookID	Topic	Title
B10	DB	Intro DB
B11	PL	More PL

1.3 Querying A Database

- SQL, the Structured Query Language, was specially designed for data processing. In contrast to Turing-complete languages such as Java, SQL initially was not intended for complex calculations, but for easy efficient access to large data. Stated differently, it could not do everything, but excelled at what it was designed for. (Note that the latest SQL versions are now Turing-complete in order to satisfy demand by some important DBMS customers.)
- SQL is based on a few comparably simple operators that can be composed into complex queries. This enables automatic query optimization.

2. Relational Database Foundations

- The mathematical foundations for databases are the relational calculus, which forms the theoretical basis for SQL, and the relational algebra, which is useful for representing query plans.

2.1 Relational Algebra

- The basic relational algebra only has five operators:
  - Selection (σ): selects a subset of rows
  - Projection (π): selects a subset of columns
  - Cross-product ($\times$): combines two relations
  - Set-difference ($-$): selects tuples that occur in one relation but not the other
  - Union ($\cup$): computes the set union
- Other operators were introduced for convenience, but do not add computational capabilities. Examples are intersection, join, set division, and renaming.
- The basic algebra is extended by an aggregation operator to compute aggregates such as sum, count, or max for a given relation.
- The relational algebra is closed, i.e., each operator works on relations and returns a relation. This allows the operators to be composed into complex queries.

2.1.1 Example for Selection

- $\sigma_{\text{Age}>25}(\text{Students})$

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	20	3.9

SID	Name	Age	GPA
2	Bob	27	3.4

2.1.1 Example for Projection

- $\pi_{\text{Name,Age}}(\text{Students})$

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	20	3.9

Name	Age
Alice	18
Bob	27
Carla	20
Dan	20

2.1.1 Examples for Union, Intersection, and Set-Difference

- For these operators to be well-defined, the input relations have to be union-compatible. This means that participating relations have to have the same schema.

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	20	3.9

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
5	Erin	19	3.6
6	Frank	20	3.8

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	20	3.9
5	Erin	19	3.6
6	Frank	20	3.8

$S1 \cup S2$

SID	Name	Age	GPA
3	Carla	20	3.8
4	Dan	20	3.9

$S1 - S2$

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4

$S1 \cap S2$

2.1.1 Example for Cross-Product

- The cross-product pairs up each tuple from one input relation with each tuple from the other. To be formally correct, it would need to rename those attributes occurring in both. (In the example the common SID attribute is therefore shown in parentheses in the result.)

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8

SID	BookID	Date
2	B10	01/17/12
3	B11	01/18/12

(SID)	Name	Age	GPA	(SID)	BookID	Date
1	Alice	18	3.5	2	B10	01/17/12
1	Alice	18	3.5	3	B11	01/18/12
2	Bob	27	3.4	2	B10	01/17/12
2	Bob	27	3.4	3	B11	01/18/12
3	Carla	20	3.8	2	B10	01/17/12
3	Carla	20	3.8	3	B11	01/18/12

$S \times R$

2.1.1 Example for Joins

- The cross-product is rarely used in practice, because (i) it is very expensive to compute and (ii) its result is often not meaningful. In the example, it intuitively does not make sense to combine Alice's student tuple with the book reservation made by Bob.
- In practice, users are typically interested in some subset of the cross-product. This is referred to as a condition-join or theta-join. More formally, given a condition C, the theta-join is the subset of cross-product tuples that satisfy C:
  - $R \bowtie_C S = \sigma_C(R \times S)$
- Here $\bowtie$ denotes the join operator. The most common join types in practice are equi-joins and natural joins. A join is an equi-join if all constraints in C are equality-conditions. The natural join is an equi-join on all the attributes that the two joined relations have in common.
  - In the equi-join example below, the duplicate SID column is omitted, because by definition the values have to be identical for an equi-join.
  - Note how the tuples in the equi-join result have real meaning. Intuitively the join added the detailed student information to each reservation tuple.

SID	Name	Age	GPA	BookID	Date
2	Bob	27	3.4	B10	01/17/12
3	Carla	20	3.8	B11	01/18/12

$$R \bowtie_{R.SID=S.SID} S$$

2.1.2 More Complex Query Example

- Let's see how we can find the names of all students who reserved a DB book. Student names appear in the Students table (S), book topics is the Books table (B), and reservation information in the Reservations table (R). Hence all three have to be joined together, using the natural join. From the joined relation, the relevant tuples are found using the corresponding selections:
  - $\pi_{\text{Name}}\left(\sigma_{\text{Topic=DB}}(B \bowtie R \bowtie S)\right)$
- The same result is obtained by the following expression:
  - $\pi_{\text{Name}}\left(\pi_{\text{SID}}\left(\left(\pi_{\text{BookID}}\sigma_{\text{Topic=DB}}B\right) \bowtie R\right) \bowtie S\right)$
- [Challenge question: Since both expressions are equivalent, which one should the database use?]

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8

SID	BookID	Date
2	B10	01/17/12
3	B11	01/18/12

BookID	Topic	Title
B10	DB	Intro DB
B11	PL	More PL

2.2 Relational Calculus

- Relational algebra specifies in exactly which order the operators are executed, i.e., HOW to compute the query result. The relational calculus, which forms the basis of the SQL query language, specifies WHAT to compute.
- Stated differently, the relational algebra is not declarative, but the relational calculus is. Usually many possible different, but equivalent, algebra "implementations" exist for a given calculus expression.
- There are two types of relational calculus:
  - In the tuple relational calculus (TRC), variables range over tuples.
  - In the domain relational calculus (DRC), variables ranges over attribute values.
- Calculus expressions are called formulas. An answer tuple is an assignment of constants to variables that make the formula evaluate to *true.*

2.2.1 Domain Relational Calculus (DRC)

- In DRC, a query has the general form $\{\langle x_1, x_2, \ldots, x_n\rangle \mid p(x_1, x_2, \ldots, x_n)\}$. This is the set of all tuples $\langle x_1, x_2, \ldots, x_n\rangle$ that make formula $p(x_1, x_2, \ldots, x_n)$ evaluate to true. Hence by specifying conditions in formula $p(\ldots)$, the user specifies the result she is interested in.
- All variables $x_1, x_2, \ldots, x_n$ must be free, i.e., not bound by a quantifier, in formula $p(\ldots)$. [Link: Explain this a little more.] No other variable in $p(\ldots)$ is allowed to be free.
- A formula in DRC is defined as follows: (Here op is a comparison operation, i.e., one of $<, >, =, \neq, \leq, \geq$.)
  - An atomic formula is a DRC formula. Possible atomic formulas are
    - $\langle x_1, x_2, \ldots, x_n\rangle \in \text{Relation}$: requires the tuple to occur in the relation.
    - x op y: compares two attribute values.
    - x op const: compares an attribute value to a constant.
  - If p and q are DRC formulas, then so are
    - $\neg p, p \wedge q, p \vee q$, and
    - $\exists x(p(x))$ and $\forall x(p(x))$ where variable x is free in $p(x)$.

2.2.2 DRC Query Example

- The following DRC query finds all students with GPA above 3.6:
  - $\{\langle S, N, A, G\rangle \mid \langle S, N, A, G\rangle \in \text{Students} \wedge G > 3.6\}$
- The first condition ensures that domain variables S, N, A, and G have to be attributes of the same student tuple. Without this condition, they could take on arbitrary values that might not correspond to any existing student.
- The second condition enforces the GPA requirement.

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	20	3.9

2.2.2 DRC Example With Join

- Consider a query to find all students with GPA above 3.6 who reserved book B10:
  - $\{\langle S, N, A, G\rangle \mid \langle S, N, A, G\rangle \in \text{Students} \wedge G > 3.6 \wedge \exists S_2, B, D(\langle S_2, B, D\rangle \in \text{Reservations} \wedge S = S_2 \wedge B = \text{B10})\}$
- For readability, the formula uses $\exists S_2, B, D$ as a shorthand for $\exists S_2 \left(\exists B\left(\exists D(\ldots)\right)\right)$.
- Intuitively, the DRC formula specifies the following conditions:
  - S, N, A, and G are attributes of a student tuple whose GPA is above 3.6.
  - There exists a reservations tuple whose SID value (S_2) is identical to the SID of the student tuple (S). This specifies the equi-join between Students and Reservations.
  - And furthermore, the BookID of the reservation tuple (B) is equal to B10.

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8

SID	BookID	Date
2	B10	01/17/12
3	B11	01/18/12

2.3 Safe Queries and Expressive Power

- We have seen two formalisms for expressing queries: relational algebra and (domain) relational calculus. How do they compare in terms of their expressive power? Can one of them express queries that the other cannot?
- Before answering this question, let's introduce the notion of a safe query:
  - It is possible to write calculus queries with an infinite number of answers. These are called unsafe queries and they are not useful in practice. Given fixed input relations, a safe query is one that always returns the same result, no matter the attribute domains. For example, $\{\langle S, N, A, G\rangle \mid (\langle S, N, A, G\rangle \in \text{Students})\}$ is safe, because S, N, A, and G have to be selected from the existing tuples in Students. On the other hand, $\{\langle S, N, A, G\rangle \mid \neg(\langle S, N, A, G\rangle \in \text{Students})\}$ is unsafe, because S, N, A, and G can take on any value in their respective domains as long as they do not match any of the (finitely many) tuples in Students.
- It has been shown that every relational algebra query can be expressed as a safe query in relational calculus, and vice versa.
- Furthermore, SQL, the programming language used to query a DBMS in practice, can express every relational algebra query. This property is called "relational completeness."

3. SQL

- Database queries are written in SQL.

3.1 Basic SQL Query

SELECT	[DISTINCT] attribute-list
FROM	relation-list
WHERE	condition

SELECT	DISTINCT Name
FROM	Students S, Reservations R
WHERE	S.SID = R.SID AND BookID = ‘B10’

- The basic SQL query consists of SELECT, FROM, and WHERE clauses.
- SELECT corresponds to the projection operator, eliminating all attributes except those in attribute-list from the result. Only attributes occurring in one of the relations in relation-list can be specified.
- FROM specifies in relation-list the relation names, possibly with a range-variable after the name, of all relations involved in the query. The same relation might appear multiple times.
- The WHERE clause specifies conditions involving comparisons of attributes or attributes against constants, using $<, >, =, \neq, \leq, \geq$.
- The DISTINCT keyword specifies that all duplicates should be eliminated from the query result. By default, an SQL query will not eliminate duplicates, because this is a costly operation.
- The semantic of the basic SQL query is defined in terms of the following conceptual evaluation strategy:
  1. Compute the cross-product of all relations in relation-list.
  2. Discard all tuples that fail the WHERE condition.
  3. Remove all attributes that are not in attribute-list.
  4. If DISTINCT is specified, remove all duplicate tuples.
- Note that this strategy will usually not be very efficient in practice. In particular, it would be wasteful to compute the full cross-product first and then eliminate tuples and attributes. The task of the database optimizer is to automatically find a better evaluation plan while guaranteeing the same result.

3.1.1 Example With Two Joins

Find all students who reserved a DB or PL book:

SQL query:

```
SELECT S.SID
FROM  Students S, Reservations R, Books B
WHERE S.SID = R.SID AND R.BookID = B.BookID
        AND (B.Topic = 'DB' OR B.Topic = 'PL')
```

DRC query:

$$\{\langle S, N, A, G\rangle \mid \langle S, N, A, G\rangle \in \text{Students} \wedge \exists S_2, B, D(\langle S_2, B, D\rangle \in \text{Reservations} \wedge S = S_2 \wedge \exists B_2, O, I(\langle B_2, O, I\rangle \in \text{Books} \wedge B = B_2 \wedge (O = \text{DB} \vee O = \text{PL}))\}$$

Note the similarity in structure between DRC and SQL.

3.1.2 Nested Query with Correlation

```
SELECT   S.name
FROM     Students S
WHERE    EXISTS   (SELECT *
                   FROM Reservations R
                   WHERE R.BookID = 'B10' AND S.SID = R.SID)
```

- The query finds the names of all students who reserved book B10.
- EXISTS tests if a set is empty. In this case, the inner query will return an empty result if for student S there is no reservation tuple for B10 that has the student's SID in the SID column.
- The nested query is correlated, because it references attributes, in particular S.SID, from the outer query.

3.2 Simple Aggregation in SQL

- SQL supports aggregation operators, in particular COUNT, SUM, AVG, MAX, and MIN.
- Users can add their own user-defined aggregates for use in their queries.
- Example aggregate queries:

Average GPA of students older than 20:

```
SELECT  AVG(GPA)
FROM    Students
WHERE   Age > 20
```

Total number of student tuples:

```
SELECT  COUNT(*)
FROM    Students
```

The name(s) of the student(s) with the highest GPA:

```
SELECT  S.name
FROM    Students S
WHERE   S.GPA =  (SELECT MAX(S2.GPA)
                  FROM Students S2)
```

3.3 GROUP BY and HAVING

```
SELECT      [DISTINCT] attribute-list
FROM        relation-list
WHERE       condition
GROUP BY    grouping-list
HAVING      group-condition
```

- SQL's GROUP BY clause partitions a relation on the attributes in the grouping-list. In particular, all tuples with the same values for all grouping attributes will end up in the same partition. After grouping, the individual tuples are replaced by an aggregate for the entire group.
- HAVING specifies a group-condition. Groups that do not satisfy this condition will not be in the result.
- The attribute-list in the SELECT clause contains attribute names and terms with aggregate operations on attributes.

3.3.1 Grouping Query Example

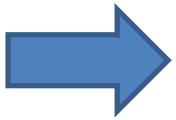
- Grouping queries are best understood through an example. The query below computes the following result: Among all students with GPA > 3.4, find the lowest GPA for each age group with at least 2 students in it.
  - The GROUP BY clause groups students by age.
  - The HAVING clause eliminates groups with fewer than 2 students.
  - Notice that the final output has schema (Age, MinGPA). Other student attributes such as Name cannot appear in the SELECT clause, because the students in an age group will have different names. Similarly, they will have different ages, therefore for Age to appear in any way in the final output, it has to be aggregated. In the example, the minimum age for each group is chosen.

```
SELECT      S.Age, MIN(S.GPA) AS MinGPA
FROM        Students S
WHERE       S.GPA > 3.4
GROUP BY    S.Age
HAVING      COUNT(*) >= 2
```

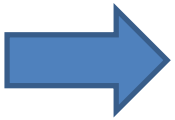
SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	20	3.9
6	Frank	20	3.8
7	Gina	27	3.8
8	Hal	18	3.5

3.3.1 Evaluation of the Grouping Query

SID	Name	Age	GPA
1	Alice	18	3.5
~~2~~	~~Bob~~	~~27~~	~~3.4~~
3	Carla	20	3.8
4	Dan	20	3.9
6	Frank	20	3.8
7	Gina	27	3.8
8	Hal	18	3.5



Age	GPA
18	3.5
20	3.8
20	3.9
20	3.8
27	3.8
18	3.5

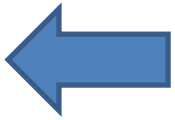


Age	GPA
18	3.5
18	3.5

Age	GPA
27	3.8

Age	GPA
20	3.8
20	3.9
20	3.8

Age	GPA
20	3.8
20	3.9
20	3.8



Age	GPA
18	3.5
18	3.5

Age	MinGPA
18	3.5
20	3.8

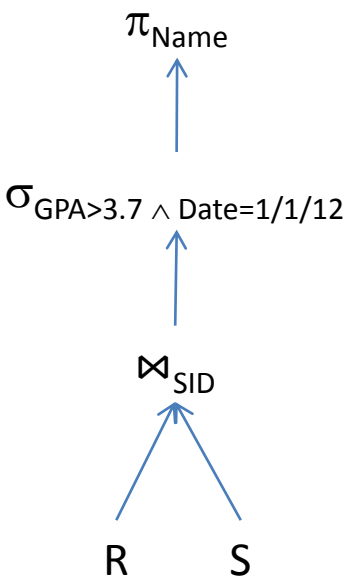
4. Query Evaluation

- How does the database optimizer choose a good execution strategy for a given SQL query?
- System-R style optimizers are the most widely used type of optimizer. They rely on principles introduced by the optimizer for System R, one of the first relational DBMS that was designed and built in the 1970s by IBM Research. It performs several optimization steps:
  - The SQL query is transformed to an initial plan consisting of multiple query blocks.
  - The optimizer then considers alternative plans by leveraging relational algebra equivalences.
  - For each plan, the combined CPU and I/O cost is estimated. This process relies on estimates of the size of intermediate results. Accurately estimating intermediate result size is a well-explored hard problem, especially for joins.
  - Due to the large search space of possible plans, the optimizer uses heuristics to enumerate promising candidate plans. For example, an optimizer would generally attempt to push selections and projections “down” to apply them as early as possible. And it would explore different join execution orders.

4.1 What Is a Query Plan?

- To reason about the cost of a query execution strategy, the optimizer represents this strategy as a query plan. A query plan is a directed acyclic graph (DAG) of relational algebra operators and their implementations.
- Operator implementation needs to be considered, because cost might vary significantly, e.g., depending on if an existing index is used for a selection operator or if the entire relation has to be scanned.
- Operators implement a pull interface, meaning that an operator calls “get next” on the output of the operators directly “upstream” from it. More precisely, if operator O2 processes the output of operator O1, then O2 will ask O1 for the next tuple. This enables pipelining between O1 and O2, avoiding the need for buffering of intermediate results.

4.2 Optimization Example



Relational-algebra tree

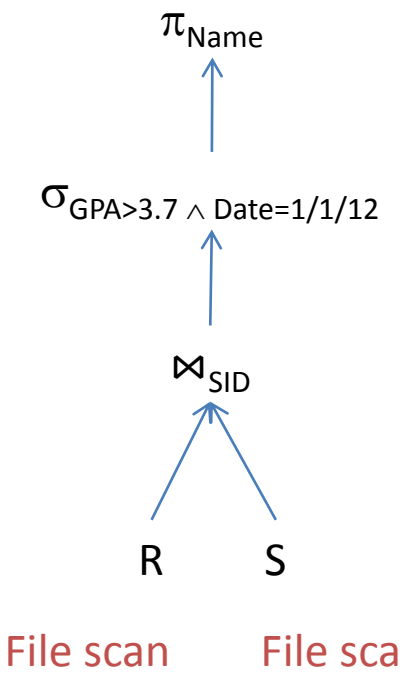
On-the-fly

On-the-fly

Block Nested Loops

Cost: 1000+1000·500 for join
plus zero I/O for on-the-fly

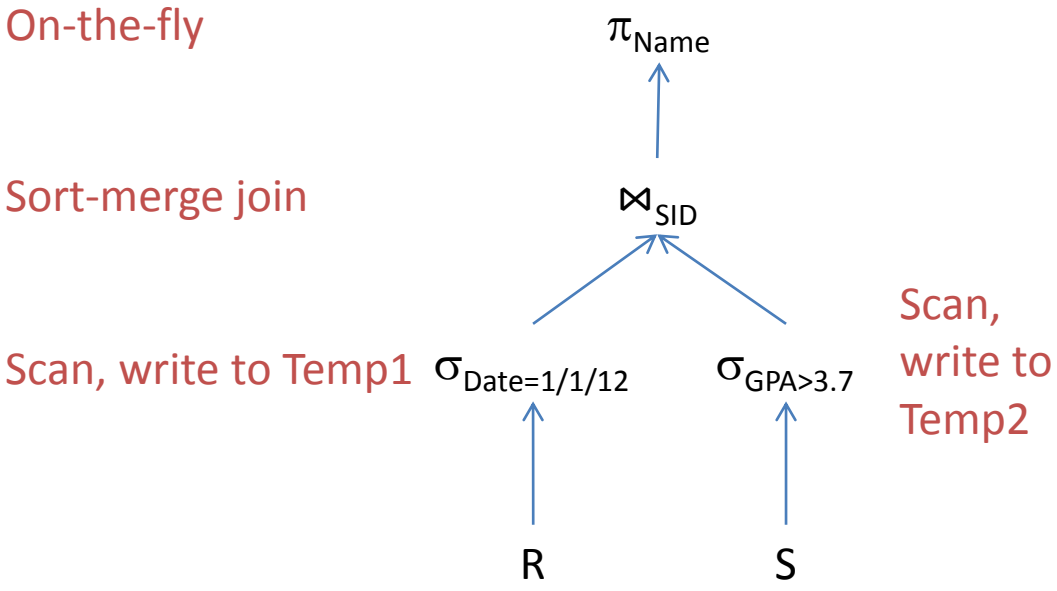
Total: ~500,000 I/O



Possible query plan

4.2 Alternative Plan: Push Selections

- Cost to scan R, write Temp1: 1000+10
- Cost to scan S, write Temp2: 500+250
- Join cost = cost for sorting + cost for merging = (2·2·10 + 2·2·250) + (10+250)
- Total cost: 3060 I/O



4.2 Another Alternative: With Indexes

- Cost to scan R: 1000
- Join cost: for each R-tuple, index lookup on S: 900·1.2
- Total: 2080 I/O

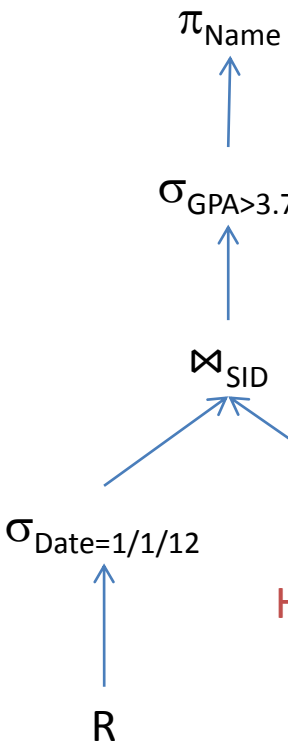
On-the-fly

On-the-fly

Index Nested Loops with pipelining

On-the-fly

File scan



Hash index on SID

Cannot push selection on GPA before the join, because that would prevent use of index

5. Parallel and Distributed Databases

- How does what we have learned so far about relational databases apply in a distributed environment where multiple machines are available to process a query?
- The beauty of SQL's declarative approach is that it can be used for writing queries in single-processor and distributed environments without any modification. Since SQL specifies what the user is looking for, a distributed database only needs a different optimizer that can generate a distributed execution plan. In addition to optimizing for total cost of a plan, in a distributed environment users might put more emphasis on optimizing for fast response time.
- Even though SQL itself (as well as relational calculus) remains the same, the distributed optimizer's notion of a *query plan* needs to be extended to take data location and network cost into account. This is typically achieved by adding new operators that deal with data transfers in query plans:
  - The Exchange operator behaves like an iterator, delivering the next tuple on request. However, it receives input via inter-process communication (from another machine) rather than iterator procedure calls.
  - The Split and Merge operators explicitly create and join parallel dataflows.

5. Parallel and Distributed Databases

- In addition to these new operators, distributed databases also benefit from new implementations for existing relational operators, e.g., semi-join based implementations of joins:
  - Data transfer cost for joins can be reduced though implementations relying on the semi-join. For example, assume relation R is on machine M1 and relation S is on a different machine M2. Computing the join of R and S on the SID attribute would require sending R to M2 or S to M1, incurring high network cost if both relations are big. If the join *result* is small, a better strategy could be as follows:
    - M2 sends $\pi_{SID}(S)$, i.e., the SID values occurring in S, to M1.
    - M1 determines the set of all R-tuples whose SID matches any of the SIDs received from M2. This is done with a semi-join.
    - M1 then sends only those R-tuples to M2. By not sending R-tuples that would not have a match in S anyway, network traffic can be reduced significantly, depending on the data distribution.
    - M2 finally performs the actual join.

5.1 Distributed Query Optimization

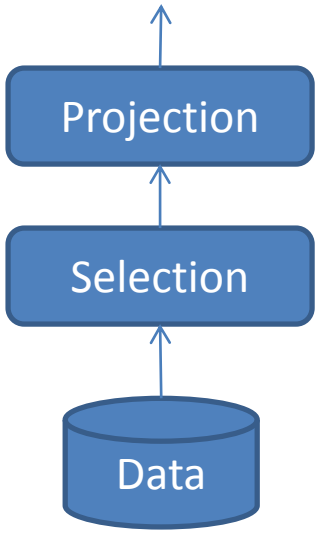
- Given an SQL query, the distributed-database optimizer works similar to a classic (non-distributed) one. The main difference is that new operators and implementations need to be considered for the query plans, taking data location and network cost into account. The optimizer performs the following steps:
  1. Start with an SQL (i.e., relational calculus) query on global relations.
  2. Transform it into a relational algebra expression, still on global relations. (Up to here the optimizer did not take into account that a relation might be split into many chunks and distributed over multiple locations.)
  3. Perform data localization, using the fragment schema, to generate an algebraic query on the relation fragments. The fragment schema specifies where the different splits of a relation are located.
  4. Perform global optimization to create a distributed query execution plan. This plan would explicitly specify necessary data transfers, e.g., using Exchange operators.
  5. Run the execution plan on all local sites in parallel.

5.2 Parallelism in Databases

- There are two types of parallelism: pipeline parallelism and partitioned parallelism.

5.2.1 Pipeline Parallelism

- Consider a pipeline where each operator pulls its next input tuple from the output of the previous operator. If multiple CPUs are available, a different CPU can be assigned to each operator in the pipeline. Similar to an assembly line, the CPUs work in parallel, moving tuples through the pipeline.



CPU2 pulls tuples from the selection operator.

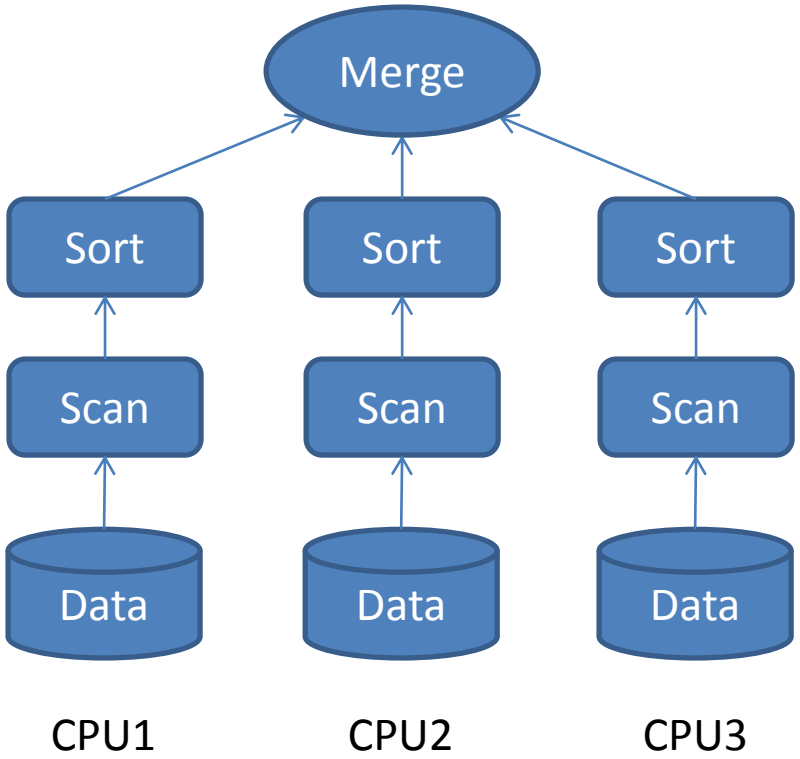
CPU1 pulls tuples from the data file.

5.2.1 Limitations of Pipeline Parallelism

- Relational pipelines are usually not very long, rarely exceeding ten operators.
- Even for complex queries, blocking operators [Link: What does this mean?] such as aggregation and sorting cannot be pipelined and hence break an otherwise long pipeline into shorter pieces.
- Execution cost might vary significantly between operators in a pipeline; and the slowest operator determines the speed of the entire pipeline. Hence if one operator is significantly slower than the others, the other CPUs will be mostly idle, waiting for the slow operator. This inherently limits speedup and scaleup of pipelining.

5.2.2 Partitioned Parallelism

- When a query performs some batch-style computation on many input tuples, partitioned parallelism is achieved by splitting the input data and having a different CPU work on each of them independently. This divide-and-conquer style of parallelism corresponds to the approach taken by MapReduce.



5.2.2 How to Partition the Data?

- Depending on the problem, the best way of partitioning a given input relation will vary. For databases, traditionally there are three major partitioning types.
- Round-robin: Each tuple is assigned to a partition arbitrarily. An exact round-robin scheme would assign the i-th tuple to partition (i mod p), where p denotes the number of partitions, but that is not really necessary. In practice the main goal of this approach is to simply distribute tuples evenly.
  - Strengths:
    - It is very easy to distribute a relation into any number of approximately equal-sized partitions. Even if tuples are inserted and deleted frequently, it is easy to maintain a well-balanced partitioning.
    - It works very well for problems where some non-trivial computation has to be applied to all (or most) input tuples.
  - Weaknesses:
    - It is not helpful for "associative access", i.e., when only a small fraction of tuples selected by some condition on their attribute values need to be processed. For example, a query processing only the 20-year old students would still have to access all partitions of the Students relation, because the 20-year old students could be in any partition.

5.2.2 How to Partition the Data?

- Hash partitioning: Each tuple is assigned to a partition based on a hash function. More precisely, tuple t is assigned to partition (h(t) mod p), where h() is a properly chosen hash function.
  - Strengths:
    - It performs well when only tuples satisfying some equality condition need to be processed. For example, assume the Students relation is hash-partitioned on age. Then there is exactly one partition that contains all 20-year olds. Hence a query processing 20-year old students only has to access that single partition.
    - Assuming hash function h produces a sufficiently large number of distinct output values for the given relation and also assigns them randomly to the different partitions, hash partitioning will result in a fairly well-balanced partitioning.
  - Weaknesses:
    - It is not helpful for range selection predicates. Consider the Students relation hash partitioned on age into p=5 partitions. A query processing students who are between 20 and 27 years old will access 8 different age groups, which are approximately randomly distributed over the different partitions. Hence there is a high probability that all partitions would need to be accessed.
    - For skewed data, it is likely that some partitions will be significantly larger than others.
- Range partitioning: Tuples are assigned to partitions based on continuous ranges. For instance, for p=4 Students might be partitioned by assigning age range [0,19] to partition 0, [20,29] to partition 1, [30,39] to partition 2, and all older students to partition 3.
  - Strengths:
    - It performs well when only tuples satisfying some range or equality condition need to be processed. For example, the query processing 20 to 27-year olds would only access partition 1.
    - It performs well for parallel sort. Each partition can be sorted independently. Then a simple concatenation suffices to obtain the entire sorted relation.
  - Weaknesses:
    - Range boundaries have to be chosen carefully based on data distribution and expected query access patterns. Otherwise this scheme risks data skew (uneven partitions) and execution skew (uneven access pattern). For instance, the vast majority of US undergraduate students tends to be in the age bracket between 18 and 22 years. Furthermore, if tuples are frequently inserted and deleted, ideal boundaries for range partitioning might shift, requiring potentially expensive re-partitioning.

6. Relational Databases vs. MapReduce

- Distributed databases and MapReduce rely on the same principles to achieve scalability for big data processing: partition the data and distribute it over many machines, then apply the same batch computation in parallel to each partition.
- The query languages—declarative SQL with optimizer versus procedural Java—are obviously different. However, these differences concern the *interface* between user and data-processing backend, not the backend itself. As systems such as Pig and Hive (discussed in another module) demonstrate, database-style query functionality can be added to MapReduce as well.
- This prompts the question if there are any major differences in the data-processing backend.
- Indeed there are and they have to do with in-place updates and transactions.

6.1 Updates and Transactions

- Databases were designed not only for big bulk-processing jobs, but also to support quick updates on a small fraction of the data. For instance, a database managing bank accounts would have to reflect the new account balance immediately after money is withdrawn. In a supermarket, the database has to record purchase transactions and update availability in realtime as cashiers scan the purchased items.
- MapReduce was not designed for interactive small updates. It takes seconds or minutes just to set up a MapReduce job and there is no support for in-place updates in an existing file. MapReduce also lacks the infrastructure for quickly locating a specific record or small set of records in a big file. On the other hand, databases were designed to excel for these types of problems.
- In addition to index structures such as B-tree and hash index for finding data quickly, the concept of a transaction was crucial for the success of database systems. A transaction is a user program that consists of a sequence of database read and write operations. Even though multiple transactions are processed in parallel, they let users write programs under the illusion that there is no concurrent access at all. Stated differently, the user simply has to make sure that her own transaction is correct if it was run in isolation. The DBMS then automatically takes care of scheduling and correct concurrent execution.

6.2 ACID Properties

- Transactions are so important that Jim Gray was awarded the 1998 Turing Award (considered the Nobel Price for computer science) for his contributions to their design and implementation. From a user's point of view, transactions matter because they guarantee four crucial properties that are know as the ACID properties:
- Atomicity guarantees that either all or none of the transaction's actions are executed, even in the presence of a failure or crash occurring mid-way through the execution. Hence the user does not have to worry about inconsistent data due to a "half-processed" program.
  - Consider a transaction transferring money from a customer's checking to her savings account. If a crash happens right after the money was withdrawn from checking but before it was deposited into savings, then the withdrawn money should not just disappear. It should either remain in the checking account or be completely transferred into the savings account.
- Consistency guarantees that a transaction will transition the database from one valid state to another, as long as the user makes sure that the transaction run by itself preserves consistency of the database.
- Isolation guarantees that transaction semantics do not depend on other concurrently executed transactions. Intuitively, when designing a transaction, the user only needs to worry about the transaction itself, not what other transactions in the database might be doing at the same time.
- Durability guarantees that effects of successfully committed transactions will persist, even when crashes occur.
  - Returning to the banking example, assume the user received confirmation about the successful money transfer. If she checks her account later again, that transfer's effect should still be reflected in her account. It would be undesirable if the money had magically gone back to the checking account.

6.2.1 Transaction Execution Example

```
T1:    BEGIN   A=A+100, B=B-100   END
T2:    BEGIN   A=1.01A, B=1.01B   END
```

- To understand the challenges faced by a database in guaranteeing ACID, consider a simple example of two bank accounts, A and B, owned by same user. Assume the user transfers $100 from B to A (transaction T1). At the same time the bank initiates a transaction T2 that credits 1% interest to all accounts.
- Let the accounts have initial balances A=500 and B=500.
  - If T1 is executed completely before T2, then the new account balances will be A=606 and B=404.
  - If T2 is executed completely before T1, then the new account balances will be A=605 and B=405.
  - Assuming both transactions were submitted concurrently, either serial execution order would be correct and the user receives the expected $10 in interest.

6.2.2 Interleaving Execution

Scenario 1:

T1: A=A+100, B=B-100
T2: A=1.01A, B=1.01B

Scenario 2:

T1: A=A+100, B=B-100
T2: A=1.01A, B=1.01B

Abstract view of Scenario 2:

T1: R(A), **W(A)**, **R(B)**, W(B)
T2: R(B), **W(B)**, **R(A)**, W(A)

6.2.2 Interleaving Execution

- For performance reasons, the database executes many transactions concurrently. Would all such interleaved executions produce correct results?
  - Scenario 1 produces a correct result that is equivalent to executing T1 completely before T2.
  - Scenario 2 is problematic. After both transactions complete, the new account balances are A=606 and B=505. This does not agree with either serial execution order (T1 completely before T2; or T2 completely before T1). Also, the user received $11 in total interest, which should not happen with a 1% interest rate for a total of $1000 in her accounts.
  - [Challenge question]
- To guarantee ACID, the database has to be able to distinguish between correct (Scenario 1) and incorrect (Scenario 2) concurrent execution of transactions. It does so based on an abstract view of an execution schedule as a sequence of database read and write operations. Since write operations modify data, they establish an ordering of transactions: if T1 writes A before T2 reads it, then T2 works with T1's updated data and hence the entire program T2 should conceptually happen after T1. Whenever the write operations in an interleaved schedule create inconsistent ordering constraints, the database can detect this and prevent this execution order by delaying a transaction or aborting it.
  - In Scenario 2, first T1 reads account balance A, then writes the new value to it. Next, T2 reads and writes both A and B to credit the interest. Finally T1 reads and writes B to complete the withdrawal. T1's W(A) and T2's R(A) tell the database that T1 should conceptually happen before T2. However, T2's W(B) and T1's R(B) tell the opposite order, signaling a problem.

6.3 Scheduling Transactions

- Given several transactions to be executed, how should the database schedule them?
- It could use a completely serial schedule, i.e., execute one transaction completely before executing the next. By not interleaving the actions of different transactions, it is easy to achieve consistency and isolation. Unfortunately, serial execution results in poor throughput. As one transaction stalls waiting for data to be brought in from disk, another could use the CPU to perform useful work with data it already brought into memory earlier. In practice, all commercial DBMS are designed to execute multiple transactions concurrently.
- Instead of a serial schedule, a DBMS will find a serializable schedule. This is a schedule where transactions can be interleaved, i.e., executed concurrently, but its effect on the database is equivalent [Link: Explain equivalence of schedules.] to some serial execution of the transactions. Intuitively the notion of a serializable schedule retains the correctness property of a serial schedule, but addresses the performance issue.
- Since serial schedules of correct transactions by definition satisfy consistency and isolation, and since each serializable schedule is equivalent to some serial one, it follows that every serializable schedule preserves consistency and isolation.

- But how does the database ensure that it will only generate serializable schedules? We first need to take a quick look at possible anomalies and then discuss how locking can prevent them.

6.3.1 Anomalies: WR

T1: R(A), W(A), R(B), W(B), Abort
T2: R(A), W(A), C

- The WR anomaly occurs for a read following a write on the same object. It is also known as “reading uncommitted data”, “WR conflict”, or “dirty read.”
  - In the example T2 reads value A written by T1 before T1 completed all its changes. If T1 later aborts, T2 worked with invalid data.
  - Example: T1 deposits a check to A and T2 credits interest to A.
- A dirty read can result in an unrecoverable schedule. In the example, T2 commits (C action at the end) before T1 completes. Because of durability, this commit cannot be undone. However, when T1 aborts, all its updates have to be undone, creating a situation where T2 worked with invalid data and hence would have to be aborted as well. To prevent this problem, the DBMS cannot allow T2 to commit until T1 has committed.
  - T2 has to wait until it knows if T1 committed successfully. If T1 fails, T2 has to be aborted. Hence dirty reads, even if handled properly by delaying commit, can still lead to cascading aborts.

6.3.2 Anomalies: RW

T1: R(A), R(A), W(A), C
T2: R(A), W(A), C

- The RW anomaly is also known as "unrepeatable read" or "RW conflict."
- In the example, T1 sees different values of A, even though it did not change it. This violates the isolation property.

6.3.3 Anomalies: WW

T1: W(A), W(B), C
T2: W(A), W(B), C

- The WW anomaly is also known as “overwriting uncommited data” or “WW conflict.”
- In the example, after executing the schedule, T1’s B and T2’s A persist, which could not happen with serial execution, violating consistency and isolation.
- For example, assume two employees in a company are supposed to have the same salary. Let T1 set both salaries to $4000 and T2 both to $4500. The above schedule would result in A=4500 and B=4000, even though both transactions individually would have ensured equal salary for the two.

6.4 Preventing Anomalies Through Locking

- Databases rely on locking protocols, which automatically determine when to lock and unlock the appropriate database objects [Link: What are database objects?] to allow maximum concurrency without suffering from anomalies. Different types of locks, particularly the distinction between shared and exclusive ones, provides additional flexibility in achieving this goal.
- In general the purpose of a locking protocol is to block problematic concurrent actions, but allow non-conflicting ones:
  - Read-only access: There is no limit on the number of transactions concurrently reading a database object X, as long as no transaction tries to update it. Hence multiple concurrent readers should be allowed through shared locks for reading.
  - Different objects: If transaction T1 affects bank accounts A and B, while transaction T2 works on X and Y, they can perform both their reads and their writes concurrently without interfering with each other.

6.4.1 Locking Basics

- Most locking protocols follow some variation of the following principles:
  - Before being able to read an object, the transaction needs to acquire a shared lock (S-lock) on it.
  - Before being able to modify an object, the transaction needs to acquire an exclusive lock (X-lock) on it.
  - Multiple transactions can hold a shared lock on the same object.
  - At most one transaction can hold an exclusive lock on an object.

6.4.2 Two-Phase Locking

- Two-phase locking (2PL) is a classic DBMS protocol. As the name suggests, it consists of two distinct phases.
- During phase 1 the transaction acquires the locks it needs.
- During phase 2 the transaction releases its locks, but it cannot acquire new locks any more.
- Database objects can be read and modified during both phases, as long as the transaction holds the required locks.
- 2PL was proven to only produce serializable schedules, but it does not necessarily prevent dirty reads (WR anomaly).
- Strict 2PL, a specific version of the general 2PL approach, releases all its locks in the very end, i.e., only when the transaction is completed. This prevents all anomalies (RW, WR, and WW).
- 2PL protocols might cause deadlocks, but versions exist that make deadlocks a rarity in practice.
  - For a deadlock example, consider transactions T1: R(A), W(B) and T2: R(B), W(A). Assume T1 first acquires an S-lock on A. Then T2 acquires an S-lock on B. Now both are deadlocked. T1 cannot obtain the X-lock on B, because of T2's S-lock on B. Similarly, T2 cannot obtain an X-lock on A. Unfortunately, neither transaction can release its S-lock, because that would initiate phase 2, during which no new lock can be acquired. The only way to break this deadlock is by aborting one of the involved transactions.

6.4.3 Locking Granularity: The Phantom Problem

- In the discussion so far, the "objects" to be locked by a transaction were not explicitly specified. The following example highlights that it does not suffice to only lock at the level of individual tuples.
- Assume initially the youngest student in the Students table is 18 years old. Two different transactions then try to access the Students table:
  - T1 executes "SELECT MIN(age) FROM Students" to find the age of the youngest student, then waits for some time until re-issuing the same query again.
  - T2 inserts a new student with age 17.
- Assume T1 starts executing first, acquiring an S-lock on every tuple in Students. The first execution of the minimum-age query returns 18, as expected. Assume further that in-between the two query executions of T1, T2 tries to perform its insertion. Since T1 holds locks only on tuples currently in Students, these locks would not prevent T2 from inserting a *new* tuple. Even with Strict 2PL, T2 can then release its X-lock on the newly inserted student tuple. Because of this insertion, the next execution of the minimum-age query in T1 would now return 17. This behavior of T1 violates the ACID properties. [Link: Why?]
- This is an example of the "phantom problem", because the 17-year old student appears seemingly out of nowhere in the middle of T1's execution. How can this problem be prevented?
  - T1 cannot lock a tuple that T2 will insert, but it could lock the entire Students table. Hence it would have to acquire an S-lock at a coarser granularity. An insert operation similarly would have to acquire an X-lock on the table, not on individual tuples.
- What if T1 computed a slightly different query such as "SELECT MIN(age) FROM Students WHERE GPA > 3.9"? Now locking the entire Students table seems excessive, because inserting a new student with GPA $\leq$ 3.9 would not create a problem. Advanced relational databases can allow T1 to *lock the predicate* [GPA > 3.9] on Students, instead of the entire table.
- In general, given a transaction, the DBMS will automatically select the appropriate granularity for locking. The less it locks, the better the performance.

6.4.4 Aborting a Transaction

- If a transaction T1 is aborted, all its actions have to be undone.
- If another transaction T2 has read an object written by T1, T2 must be aborted as well. This in turn might result in yet another transaction being aborted because it read objects modified by T2, and so on. Strict 2PL avoids such cascading aborts by releasing a transaction's locks only at commit time.
- 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 transactions that were active at the time of the crash are aborted when the system comes back up.

6.4.5 Performance Of Locking

- Locking adds overhead for lock management. Even if there are no conflicts, transactions have to acquire and release locks.
- In case of a conflict, locking forces transactions to wait. This is necessary for ensuring ACID, but negatively affects performance.
- Abort and restart due to deadlocks wastes the work done by the aborted transactions.
- As a tendency, the more concurrent transactions are executed, the greater the risk of *lock contention*. Allowing more concurrent transactions initially increases throughput, but at some point leads to thrashing. (Thrashing occurs when an excessive number of active transactions overwhelms system resources and individual transactions make only very slow progress because they are constantly waiting for resources to become available.) Databases therefore typically limit the number of concurrent transactions allowed.

6.4.6 Consistency and Isolation versus Performance

- Database researchers and system designers recognized early on that strong consistency guarantees come at a high cost in terms of performance. Hence relational databases allow the programmer to trade off concurrency (and hence performance) against exposure of a transaction to other transactions' uncommitted changes.
- In particular, the SQL standard enables each individual transaction to choose from four possible isolation levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The table below summarizes the consistency guarantees for a transaction depending on its selected isolation level. Performance tends to increase from bottom to top, i.e., SERIALIZABLE results in the lowest, and READ UNCOMMITTED in the highest performance.
- Weaker isolation levels are achieved by the transaction not acquiring certain locks or by releasing its read locks early. This does not affect the isolation levels selected for other transactions.
  - For example, consider isolation level READ UNCOMMITTED. It is only allowed for transactions that do not perform any kind of update. This transaction will not request any locks at all, but simply start reading whatever object it is requesting. While this transaction might see inconsistent data, it does not have to wait for any locks and does not add to lock contention in the system. [Link: Why would the user want to get potentially inconsistent results?]

Isolation Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	No	Possible	Possible
REPEATABLE READ	No	No	Possible
SERIALIZABLE	No	No	No

6.5 Distributed Transactions

- In a distributed environment, transactions take longer to access remote objects. Hence they need to hold locks longer, which increases the probability of waiting and of deadlocks.
- If the network partitions or machines are slow to respond, a transaction might not be able to acquire or release some locks.
- All machines participating in a distributed transaction have to coordinate to guarantee ACID. A machine might crash or become unreachable any time during this process.
- The most well-known standard protocol for ensuring ACID in a distributed DBMS is the two-phase commit protocol (2PC). [Alert: Be careful to not confuse this with the two-phase locking protocol (2PL)!]

6.5.1 Two-Phase Commit Overview

- The 2PC protocol distinguishes between a coordinator and multiple participants. There is only one coordinator process and it manages the communication required for agreeing on the outcome of a distributed transaction. All processes involved in the distributed transaction are participants, who communicate with the coordinator to make a global decision about the transaction's success.
- Phase 1: commit-request phase
  - The coordinator asks all participants to prepare for commit.
  - Each participant votes YES or NO to the commit request, sending the response to the coordinator.
- Phase 2: commit phase
  - Based on all participants' votes, the coordinator decides to commit if all voted YES; otherwise it will decide to abort. Missing votes will be interpreted as a NO vote.
  - The coordinator notifies all participants about its decision.
  - Each participant applies the corresponding action (commit or abort) locally.

6.5.2 Two-Phase Commit Challenges

- 2PC is a blocking protocol, because participants cannot make a decision without hearing from the coordinator. In particular, they will hold on to locks if the coordinator is down and they answered YES to the first request.
- For transactions involving many machines, the probability of aborting a transaction due to a single failed or slow-responding participant can be significant.
- Some of these issues were addressed by modifications and extensions of 2PC, but the basic problems remain. It is inherently difficult to guarantee ACID in a system with hundreds of machines participating in a distributed transaction. This will be discussed in more detail for the CAP theorem in another module.

7. Summary: Relational DBMS vs. MapReduce

	DBMS	MapReduce
Supported workload	Bulk and random reads/writes; indexes for fast access to small amounts of relevant data; transactions (ACID) can update data in existing tables	Bulk reads/writes only; no file updates (output is written in bulk to a new file)
Data representation	The DBMS controls the data format. A database schema must be specified in advance.	Any serializable Java objects can be used in Hadoop.
Programming style	Declarative (specify WHAT you want); optimizer determines HOW to compute it efficiently	Imperative (specify HOW to compute it, step-by-step)
Scalability	Limited due to ACID guarantee, requiring coordination between machines participating in a transaction	Extremely scalable due to absence of concurrent updates

Recall the discussion about Pig vs. DBMS.