Lecture #01: Relational Model & Relational Algebra

15-445/645 Database Systems (Fall 2021)

https://15445.courses.cs.cmu.edu/fall2021/

Carnegie Mellon University
Andrew Crotty

1 Databases

A *database* is an organized collection of inter-related data that models some aspect of the real-world (e.g., modeling the students in a class or a digital music store). People often confuse "databases" with "database management systems" (e.g., MySQL, Oracle, MongoDB). A database management system (DBMS) is the software that manages a database.

Consider a database that models a digital music store (e.g., Spotify). Let the database hold information about the artists and which albums those artists have released.

2 Flat File Strawman

Database is stored as comma-separated value (CSV) files that the DBMS manages. Each entity will be stored in its own file. The application has to parse files each time it wants to read or update records. Each entity has its own set of attributes, so in each file, different records are delimited by new lines, while each of the corresponding attributes within a record are delimited by a comma.

Keeping along with the digital music store example, there would be two files: one for artist and the other for album. An artist could have a name, year, and country attributes, while an album has name, artist and year attributes.

Issues with Flat File

- **Data Integrity**
  - How do we ensure that the artist is the same for each album entry?
  - What if somebody overwrites the album year with an invalid string?
  - How do we treat multiple artists on one album?
  - What happens when we delete an artist with an album?
- **Implementation**
  - How do we find a particular record?
  - What if we now want to create a new application that uses the same database?
  - What if two threads try to write to the same file at the same time?
- **Durability**
  - What if the machine crashes while our program is updating a record?
  - What if we want to replicate the database on multiple machines for high availability?

3 Database Management System

A *DBMS* is a software that allows applications to store and analyze information in a database.

A general-purpose DBMS is designed to allow the definition, creation, querying, updation, and administration of databases.

Early DBMSs

Database applications were difficult to build and maintain because there was a tight coupling between logical and physical layers. The logical layer describes which entities and attributes the database has while the physical layer is how those entities and attributes are being stored. Early on, the physical layer was defined in the application code, so if we wanted to change the physical layer the application was using, we would have to change all of the code to match the new physical layer.

4 Relational Model

Ted Codd noticed that people were rewriting DBMSs every time they wanted to change the physical layer, so in 1970 he proposed the relational model to avoid this. This relational model has three key points:

- Store database in simple data structures (relations).
- Access data through high-level language.
- Physical storage left up to implementation.

A *data model* is a collection of concepts for describing the data in a database. The relational model is an example of a data model.

A *schema* is a description of a particular collection of data, using a given data model.

The relational data model defines three concepts:

- **Structure:** The definition of relations and their contents. This is the attributes the relations have and the values that those attributes can hold.
- **Integrity:** Ensure the database's contents satisfy constraints. An example constraint would be that any value for the year attribute has to be a number.
- **Manipulation:** How to access and modify a database's contents.

A *relation* is an unordered set that contains the relationship of attributes that represent entities. Since the relationships are unordered, the DBMS can store them in any way it wants, allowing for optimization.

A *tuple* is a set of attribute values (also known as its *domain*) in the relation. Originally, values had to be atomic or scalar, but now values can also be lists or nested data structures. Every attribute can be a special value, NULL, which means for a given tuple the attribute is undefined.

A relation with n attributes is called an n*-ary relation*.

Keys

A relation's *primary key* uniquely identifies a single tuple. Some DBMSs automatically create an internal primary key if you do not define one. A lot of DBMSs have support for autogenerated keys so an application does not have to manually increment the keys.

A *foreign key* specifies that an attribute from one relation has to map to a tuple in another relation.

5 Data Manipulation Languages (DMLs)

A language to store and retrieve information from a database. There are two classes of languages for this:

- **Procedural:** The query specifies the (high-level) strategy the DBMS should use to find the desired result.
- **Non-Procedural (Declarative):** The query specifies only *what* data is wanted and not *how* to find it.

6 Relational Algebra

Relational Algebra is a set of fundamental operations to retrieve and manipulate tuples in a relation. Each operator takes in one or more relations as inputs, and outputs a new relation. To write queries we can "chain" these operators together to create more complex operations.

Select

Select takes in a relation and outputs a subset of the tuples from that relation that satisfy a selection predicate. The predicate acts like a filter, and we can combine multiple predicates using conjunctions and disjunctions.

Syntax: $\sigma_{\text{predicate}}(R)$.

Projection

Projection takes in a relation and outputs a relation with tuples that contain only specified attributes. You can rearrange the ordering of the attributes in the input relation as well as manipulate the values.

Syntax: $\pi_{\text{A1,A2,...,An}}(R)$.

Union

Union takes in two relations and outputs a relation that contains all tuples that appear in at least one of the input relations. Note: The two input relations have to have the exact same attributes.

Syntax: $(R \cup S)$.

Intersection

Intersection takes in two relations and outputs a relation that contains all tuples that appear in both of the input relations. Note: The two input relations have to have the exact same attributes.

Syntax: $(R \cap S)$.

Difference

Difference takes in two relations and outputs a relation that contains all tuples that appear in the first relation but not the second relation. Note: The two input relations have to have the exact same attributes.

Syntax: $(R - S)$.

Product

Product takes in two relations and outputs a relation that contains all possible combinations for tuples from the input relations.

Syntax: $(R \times S)$.

Join

Join takes in two relations and outputs a relation that contains all the tuples that are a combination of two tuples where for each attribute that the two relations share, the values for that attribute of both tuples is the same.

Syntax: $(R \bowtie S)$.

Observation

Relational algebra is a procedural language because it defines the high level-steps of how to compute a query. For example, $\sigma_{\text{b\_id=102}}(R \bowtie S)$ is saying to first do the join of R and S and then do the select, whereas $(R \bowtie (\sigma_{\text{b\_id=102}}(S)))$ will do the select on S first, and then do the join. These two statements will actually produce the same answer, but if there is only 1 tuple in S with b_id=102 out of a billion tuples, then $(R \bowtie (\sigma_{\text{b\_id=102}}(S)))$ will be significantly faster than $\sigma_{\text{b\_id=102}}(R \bowtie S)$.

A better approach is to say the result you want, and let the DBMS decide the steps it wants to take to compute the query. SQL will do exactly this, and it is the de facto standard for writing queries on relational model databases.

Lecture #02: Intermediate SQL

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Andrew Crotty

1 Relational Languages

Edgar Codd published a major paper on relational models in the early 1970s. Originally, he only defined the mathematical notation for how a DBMS could execute queries on a relational model DBMS.

The user only needs to specify the result that they want using a declarative language (i.e., SQL). The DBMS is responsible for determining the most efficient plan to produce that answer.

Relational algebra is based on **sets** (unordered, no duplicates). SQL is based on **bags** (unordered, allows duplicates).

2 SQL History

Declarative query lanaguage for relational databases. It was originally developed in the 1970s as part of the IBM **System R** project. IBM originally called it "SEQUEL" (Structured English Query Language). The name changed in the 1980s to just "SQL" (Structured Query Language).

The language is comprised of different classes of commands:

1. **Data Manipulation Language (DML):** `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.
2. **Data Definition Language (DDL):** Schema definitions for tables, indexes, views, and other objects.
3. **Data Control Language (DCL):** Security, access controls.

SQL is not a dead language. It is being updated with new features every couple of years. SQL-92 is the minimum that a DBMS has to support to claim they support SQL. Each vendor follows the standard to a certain degree but there are many proprietary extensions.

Some of the major updates released with each new edition of the SQL standard are shown below.

- **SQL:1999** Regular expressions, Triggers
- **SQL:2003** XML, Windows, Sequences
- **SQL:2008** Truncation, Fancy sorting
- **SQL:2011** Temporal DBs, Pipelined DML
- **SQL:2016** JSON, Polymorphic tables

3 Joins

Combines columns from one or more tables and produces a new table. Used to express queries that involve data that spans multiple tables.

Example: *Which students got an A in 15-721?*

```sql
CREATE TABLE student (
   sid   INT PRIMARY KEY,
   name  VARCHAR(16),
   login VARCHAR(32) UNIQUE,
   age   SMALLINT,
   gpa   FLOAT
);

CREATE TABLE course (
   cid  VARCHAR(32) PRIMARY KEY,
   name VARCHAR(32) NOT NULL
);

CREATE TABLE enrolled (
   sid   INT REFERENCES student (sid),
   cid   VARCHAR(32) REFERENCES course (cid),
   grade CHAR(1)
);
```

Figure 1: Example database used for lecture

```sql
SELECT s.name
  FROM enrolled AS e, student AS s
 WHERE e.grade = 'A' AND e.cid = '15-721'
   AND e.sid = s.sid;
```

4 Aggregates

An aggregation function takes in a bag of tuples as its input and then produces a single scalar value as its output. Aggregate functions can (almost) only be used in a `SELECT` output list.

- `AVG(COL)`: The average of the values in `COL`
- `MIN(COL)`: The minimum value in `COL`
- `MAX(COL)`: The maximum value in `COL`
- `COUNT(COL)`: The number of tuples in the relation

Example: *Get # of students with a '@cs' login.*

The following three queries are equivalent:

```sql
SELECT COUNT(*) FROM student WHERE login LIKE '%@cs';
```

```sql
SELECT COUNT(login) FROM student WHERE login LIKE '%@cs';
```

```sql
SELECT COUNT(1) FROM student WHERE login LIKE '%@cs';
```

Can use multiple aggregates within a single `SELECT` statement:

```sql
SELECT AVG(gpa), COUNT(sid)
  FROM student WHERE login LIKE '%@cs';
```

Some aggregate functions support the DISTINCT keyword:

```
SELECT COUNT(DISTINCT login)
  FROM student WHERE login LIKE '%@cs';
```

Output of other columns outside of an aggregate is undefined (e.cid is undefined below).

Example: *Get the average GPA of students in each course.*

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid;
```

Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid;
```

The HAVING clause filters output results based on aggregation computation. This make HAVING behave like a WHERE clause for a GROUP BY.

Example: *Get the set of courses in which the average student GPA is greater than 3.9.*

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
HAVING avg_gpa > 3.9;
```

The above query syntax is supported by many major database systems, but is not compliant with the SQL standard. To make the query standard compliant, we must repeat use of AVG(S.GPA) in the body of the HAVING clause.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
HAVING AVG(s.gpa) > 3.9;
```

5 String Operations

The SQL standard says that strings are **case sensitive** and **single-quotes only**. There are functions to manipulate strings that can be used in any part of a query.

Pattern Matching: The LIKE keyword is used for string matching in predicates.

- "%" matches any substrings (including empty).
- "_" matches any one character.

Concatenation: Two vertical bars ("||") will concatenate two or more strings together into a single string.

String Functions SQL-92 defines string functions. Many database systems implement other functions in addition to those in the standard. Examples of standard string functions include `SUBSTRING(S, B, E)` and `UPPER(S)`.

6 Date and Time

Operations to manipulate `DATE` and `TIME` attributes. Can be used in either output or predicates. The specific syntax for date and time operations varies wildly across systems.

7 Output Redirection

Instead of having the result a query returned to the client (e.g., terminal), you can tell the DBMS to store the results into another table. You can then access this data in subsequent queries.

- **New Table:** Store the output of the query into a new (permanent) table.

```
SELECT DISTINCT cid INTO CourseIds FROM enrolled;
```

- **Existing Table:** Store the output of the query into a table that already exists in the database. The target table must have the same number of columns with the same types as the target table, but the names of the columns in the output query do not have to match.

```
INSERT INTO CourseIds (SELECT DISTINCT cid FROM enrolled);
```

8 Output Control

Since results SQL are unordered, we must use the `ORDER BY` clause to impose a sort on tuples:

```
SELECT sid, grade FROM enrolled WHERE cid = '15-721'
 ORDER BY grade;
```

The default sort order is ascending (`ASC`). We can manually specify `DESC` to reverse the order:

```
SELECT sid, grade FROM enrolled WHERE cid = '15-721'
 ORDER BY grade DESC;
```

We can use multiple `ORDER BY` clauses to break ties or do more complex sorting:

```
SELECT sid, grade FROM enrolled WHERE cid = '15-721'
 ORDER BY grade DESC, sid ASC;
```

We can also use any arbitrary expression in the `ORDER BY` clause:

```
SELECT sid FROM enrolled WHERE cid = '15-721'
 ORDER BY UPPER(grade) DESC, sid + 1 ASC;
```

By default, the DBMS will return all of the tuples produced by the query. We can use the `LIMIT` clause to restrict the number of result tuples:

```
SELECT sid, name FROM student WHERE login LIKE '%@cs'
 LIMIT 10;
```

We can also provide an offset to return a range in the results:

```
SELECT sid, name FROM student WHERE login LIKE '%@cs'
 LIMIT 10 OFFSET 20;
```

Unless we use an `ORDER BY` clause with a `LIMIT`, the DBMS may produce different tuples in the result on each invocation of the query because the relational model does not impose an ordering.

9 Nested Queries

Invoke queries inside of other queries to execute more complex logic within a single query. Nested queries are often difficult to optimize.

The scope of outer query is included in an inner query (i.e. the inner query can access attributes from outer query), but not the other way around.

Inner queries can appear in almost any part of a query:

1. `SELECT` Output Targets:

   ```
   SELECT (SELECT 1) AS one FROM student;
   ```

2. `FROM` Clause:

   ```
   SELECT name
     FROM student AS s, (SELECT sid FROM enrolled) AS e
     WHERE s.sid = e.sid;
   ```

3. `WHERE` Clause:

   ```
   SELECT name FROM student
    WHERE sid IN ( SELECT sid FROM enrolled );
   ```

Example: *Get the names of students that are enrolled in '15-445'.*

```
SELECT name FROM student
 WHERE sid IN (
    SELECT sid FROM enrolled
    WHERE cid = '15-445'
 );
```

Note that `sid` has different scope depending on where it appears in the query.

Nested Query Results Expressions:

- `ALL`: Must satisfy expression for all rows in sub-query.
- `ANY`: Must satisfy expression for at least one row in sub-query.
- `IN`: Equivalent to `=ANY()`.
- `EXISTS`: At least one row is returned.

10 Window Functions

Performs "sliding" calculation across a set of tuples that are related. Like an aggregation but tuples are not grouped into a single output tuple.

Functions: The window function can be any of the aggregation functions that we discussed above. There are also also special window functions:

1. `ROW_NUMBER`: The number of the current row.
2. `RANK`: The order position of the current row.

Grouping: The `OVER` clause specifies how to group together tuples when computing the window function. Use `PARTITION BY` to specify group.

```
SELECT cid, sid, ROW_NUMBER() OVER (PARTITION BY cid)
  FROM enrolled ORDER BY cid;
```

We can also put an `ORDER BY` within `OVER` to ensure a deterministic ordering of results even if database changes internally.

```
SELECT *, ROW_NUMBER() OVER (ORDER BY cid)
  FROM enrolled ORDER BY cid;
```

IMPORTANT: The DBMS computes `RANK` after the window function sorting, whereas it computes `ROW_NUMBER` before the sorting.

11 Common Table Expressions

Common Table Expressions (CTEs) are an alternative to windows or nested queries when writing more complex queries. They provide a way to write auxiliary statements for user in a larger query. CTEs can be thought of as a temporary table that is scoped to a single query.

The `WITH` clause binds the output of the inner query to a temporary result with that name.

Example: *Generate a CTE called* `cteName` *that contains a single tuple with a single attribute set to "1". Select all attributes from this CTE.* `cteName`.

```
WITH cteName AS (
    SELECT 1
)
SELECT * FROM cteName;
```

We can bind output columns to names before the `AS`:

```
WITH cteName (col1, col2) AS (
    SELECT 1, 2
)
SELECT col1 + col2 FROM cteName;
```

A single query may contain multiple CTE declarations:

```
WITH cte1 (col1) AS (SELECT 1), cte2 (col2) AS (SELECT 2)
SELECT * FROM cte1, cte2;
```

Adding the `RECURSIVE` keyword after `WITH` allows a CTE to reference itself. This enables the implementation of recursion in SQL queries. With recursive CTEs, SQL is provably turing-complete, implying that it is as computationally expressive as more general purpose programming languages (if a bit more cumbersome).

Example: *Print the sequence of numbers from 1 to 10.*

```
WITH RECURSIVE cteSource (counter) AS (
    ( SELECT 1 )
    UNION
    ( SELECT counter + 1 FROM cteSource
      WHERE counter < 10 )
)
SELECT * FROM cteSource;
```

Lecture #03: Database Storage (Part I)

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Andrew Crotty

1 Storage

We will focus on a "disk-oriented" DBMS architecture that assumes that the primary storage location of the database is on non-volatile disk(s).

At the top of the storage hierarchy, you have the devices that are closest to the CPU. This is the fastest storage, but it is also the smallest and most expensive. The further you get away from the CPU, the larger but slower the storage devices get. These devices also get cheaper per GB.

Volatile Devices:

- Volatile means that if you pull the power from the machine, then the data is lost.
- Volatile storage supports fast random access with byte-addressable locations. This means that the program can jump to any byte address and get the data that is there.
- For our purposes, we will always refer to this storage class as "memory."

Non-Volatile Devices:

- Non-volatile means that the storage device does not require continuous power in order for the device to retain the bits that it is storing.
- It is also block/page addressable. This means that in order to read a value at a particular offset, the program first has to load the 4 KB page into memory that holds the value the program wants to read.
- Non-volatile storage is traditionally better at sequential access (reading multiple contiguous chunks of data at the same time).
- We will refer to this as "disk." We will not make a (major) distinction between solid-state storage (SSD) and spinning hard drives (HDD).

There is also a relatively new class of storage devices that are becoming more popular called *persistent memory*. These devices are designed to be the best of both worlds: almost as fast as DRAM with the persistence of disk. We will not cover these devices in this course, and they are currently not in widespread production use. Note that you may see older references to persistent memory as "non-volatile memory".

You may see references to NVMe SSDs, where NVMe stands for non-volatile memory express. These NVMe SSDs are not the same hardware as persistent memory modules. Rather, they are typical NAND flash drives that connect over an improved hardware interface. This improved hardware interface allows for much faster transfers, which leverages improvements in NAND flash perfomance.

Since our DBMS architecture assumes that the database is stored on disk, the components of the DBMS are responsible for figuring out how to move data between non-volatile disk and volatile memory since the system cannot operate on the data directly on disk.

We will focus on hiding the latency of the disk rather than optimizations with registers and caches since getting data from disk is so slow. If reading data from the L1 cache reference took half a second, reading from an SSD would take 1.7 days, and reading from an HDD would take 16.5 weeks.

2 Disk-Oriented DBMS Overview

The database is all on disk, and the data in database files is organized into pages, with the first page being the directory page. To operate on the data, the DBMS needs to bring the data into memory. It does this by having a *buffer pool* that manages the data movement back and forth between disk and memory. The DBMS also has an execution engine that will execute queries. The execution engine will ask the buffer pool for a specific page, and the buffer pool will take care of bringing that page into memory and giving the execution engine a pointer to that page in memory. The buffer pool manager will ensure that the page is there while the execution engine operates on that part of memory.

3 DBMS vs. OS

A high-level design goal of the DBMS is to support databases that exceed the amount of memory available. Since reading/writing to disk is expensive, disk use must be carefully managed. We do not want large stalls from fetching something from disk to slow down everything else. We want the DBMS to be able to process other queries while it is waiting to get the data from disk.

This high-level design goal is like virtual memory, where there is a large address space and a place for the OS to bring in pages from disk.

One way to achieve this virtual memory is by using `mmap` to map the contents of a file in a process' address space, which makes the OS responsible for moving pages back and forth between disk and memory. Unfortunately, this means that if `mmap` hits a page fault, the process will be blocked.

- You never want to use `mmap` in your DBMS if you need to write.
- The DBMS (almost) always wants to control things itself and can do a better job at it since it knows more about the data being accessed and the queries being processed.
- The operating system is not your friend.

It is possible to use the OS by using:

- `madvise`: Tells the OS know when you are planning on reading certain pages.
- `mlock`: Tells the OS to not swap memory ranges out to disk.
- `msync`: Tells the OS to flush memory ranges out to disk.

We do not advise using `mmap` in a DBMS for correctness and performance reasons.

Even though the system will have functionalities that seem like something the OS can provide, having the DBMS implement these procedures itself gives it better control and performance.

4 File Storage

In its most basic form, a DBMS stores a database as files on disk. Some may use a file hierarchy, others may use a single file (e.g., SQLite).

The OS does not know anything about the contents of these files. Only the DBMS knows how to decipher their contents, since it is encoded in a way specific to the DBMS.

The DBMS's *storage manager* is responsible for managing a database's files. It represents the files as a collection of pages. It also keeps track of what data has been read and written to pages as well how much free space there is in these pages.

5 Database Pages

The DBMS organizes the database across one or more files in fixed-size blocks of data called *pages*. Pages can contain different kinds of data (tuples, indexes, etc). Most systems will not mix these types within pages. Some systems will require that pages are *self-contained*, meaning that all the information needed to read each page is on the page itself.

Each page is given a unique identifier. If the database is a single file, then the page id can just be the file offset. Most DBMSs have an indirection layer that maps a page id to a file path and offset. The upper levels of the system will ask for a specific page number. Then, the storage manager will have to turn that page number into a file and an offset to find the page.

Most DBMSs uses fixed-size pages to avoid the engineering overhead needed to support variable-sized pages. For example, with variable-size pages, deleting a page could create a hole in files that the DBMS cannot easily fill with new pages.

There are three concepts of pages in DBMS:

1. Hardware page (usually 4 KB).
2. OS page (4 KB).
3. Database page (1-16 KB).

The storage device guarantees an atomic write of the size of the hardware page. If the hardware page is 4 KB and the system tries to write 4 KB to the disk, either all 4 KB will be written, or none of it will. This means that if our database page is larger than our hardware page, the DBMS will have to take extra measures to ensure that the data gets written out safely since the program can get partway through writing a database page to disk when the system crashes.

6 Database Heap

There are a couple of ways to find the location of the page a DBMS wants on the disk, and heap file organization is one of those ways. A *heap file* is an unordered collection of pages where tuples are stored in random order.

The DBMS can locate a page on disk given a `page_id` by using a linked list of pages or a page directory.

1. **Linked List:** Header page holds pointers to a list of free pages and a list of data pages. However, if the DBMS is looking for a specific page, it has to do a sequential scan on the data page list until it finds the page it is looking for.
2. **Page Directory:** DBMS maintains special pages that track locations of data pages along with the amount of free space on each page.

7 Page Layout

Every page includes a header that records meta-data about the page's contents:

- Page size.
- Checksum.
- DBMS version.
- Transaction visibility.
- Self-containment. (Some systems like Oracle require this.)

A strawman approach to laying out data is to keep track of how many tuples the DBMS has stored in a page and then append to the end every time a new tuple is added. However, problems arise when tuples are

deleted or when tuples have variable-length attributes.

There are two main approaches to laying out data in pages: (1) slotted-pages and (2) log-structured.

Slotted Pages: Page maps slots to offsets.

- Most common approach used in DBMSs today.
- Header keeps track of the number of used slots, the offset of the starting location of the last used slot, and a slot array, which keeps track of the location of the start of each tuple.
- To add a tuple, the slot array will grow from the beginning to the end, and the data of the tuples will grow from end to the beginning. The page is considered full when the slot array and the tuple data meet.

Log-Structured: Instead of storing tuples, the DBMS only stores log records.

- Stores records to file of how the database was modified (insert, update, deletes).
- To read a record, the DBMS scans the log file backwards and "recreates" the tuple.
- Fast writes, potentially slow reads.
- Works well on append-only storage because the DBMS cannot go back and update the data.
- To avoid long reads, the DBMS can have indexes to allow it to jump to specific locations in the log. It can also periodically compact the log. (If it had a tuple and then made an update to it, it could compact it down to just inserting the updated tuple.) The issue with compaction is that the DBMS ends up with write amplification. (It re-writes the same data over and over again.)

8 Tuple Layout

A tuple is essentially a sequence of bytes. It is the DBMS's job to interpret those bytes into attribute types and values.

Tuple Header: Contains meta-data about the tuple.

- Visibility information for the DBMS's concurrency control protocol (i.e., information about which transaction created/modified that tuple).
- Bit Map for `NULL` values.
- Note that the DBMS does not need to store meta-data about the schema of the database here.

Tuple Data: Actual data for attributes.

- Attributes are typically stored in the order that you specify them when you create the table.
- Most DBMSs do not allow a tuple to exceed the size of a page.

Unique Identifier:

- Each tuple in the database is assigned a unique identifier.
- Most common: `page_id` + (`offset` or `slot`).
- An application **cannot** rely on these ids to mean anything.

Denormalized Tuple Data: If two tables are related, the DBMS can "pre-join" them, so the tables end up on the same page. This makes reads faster since the DBMS only has to load in one page rather than two separate pages. However, it makes updates more expensive since the DBMS needs more space for each tuple.

Lecture #04: Database Storage (Part II)

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Andrew Crotty

1 Data Representation

The data in a tuple is essentially just byte arrays. It is up to the DBMS to know how to interpret those bytes to derive the values for attributes. A *data representation* scheme is how a DBMS stores the bytes for a value.

There are five high level datatypes that can be stored in tuples: integers, variable-precision numbers, fixed-point precision numbers, variable length values, and dates/times.

Integers

Most DBMSs store integers using their "native" C/C++ types as specified by the IEEE-754 standard. These values are fixed length.

Examples: `INTEGER`, `BIGINT`, `SMALLINT`, `TINYINT`.

Variable Precision Numbers

These are inexact, variable-precision numeric types that use the "native" C/C++ types specified by IEEE-754 standard. These values are also fixed length.

Operations on variable-precision numbers are faster to compute than arbitrary precision numbers because the CPU can execute instructions on them directly. However, there may be rounding errors when performing computations due to the fact that some numbers cannot be represented precisely.

Examples: `FLOAT`, `REAL`.

Fixed-Point Precision Numbers

These are numeric data types with arbitrary precision and scale. They are typically stored in exact, variable-length binary representation (almost like a string) with additional meta-data that will tell the system things like the length of the data and where the decimal should be.

These data types are used when rounding errors are unacceptable, but the DBMS pays a performance penalty to get this accuracy.

Examples: `NUMERIC`, `DECIMAL`.

Variable-Length Data

These represent data types of arbitrary length. They are typically stored with a header that keeps track of the length of the string to make it easy to jump to the next value. It may also contain a checksum for the data.

Most DBMSs do not allow a tuple to exceed the size of a single page. The ones that do store the data on a special "overflow" page and have the tuple contain a reference to that page. These overflow pages can contain pointers to additional overflow pages until all the data can be stored.

Some systems will let you store these large values in an external file, and then the tuple will contain a pointer to that file. For example, if the database is storing photo information, the DBMS can store the photos in the external files rather than having them take up large amounts of space in the DBMS. One downside of this is that the DBMS cannot manipulate the contents of this file. Thus, there are no durability or transaction protections.

Examples: `VARCHAR`, `VARBINARY`, `TEXT`, `BLOB`.

Dates and Times

Representations for date/time vary for different systems. Typically, these are represented as some unit time (micro/milli)seconds since the unix epoch.

Examples: `TIME`, `DATE`, `TIMESTAMP`.

System Catalogs

In order for the DBMS to be able to deciphter the contents of tuples, it maintains an internal catalog to tell it meta-data about the databases. The meta-data will contain information about what tables and columns the databases have along with their types and the orderings of the values.

Most DBMSs store their catalog inside of themselves in the format that they use for their tables. They use special code to "bootstrap" these catalog tables.

2 Workloads

There are many different workloads for database systems. By workload, we are referring to the general nature of requests a system will have to handle. This course will focus on two types: Online Transaction Processing and Online Analytical Processing.

OLTP: Online Transaction Processing

An OLTP workload is characterized by fast, short running operations, simple queries that operate on single entity at a time, and repetitive operations. An OLTP workload will typically handle more writes than reads.

An example of an OLTP workload is the Amazon storefront. Users can add things to their cart, they can make purchases, but the actions only affect their account.

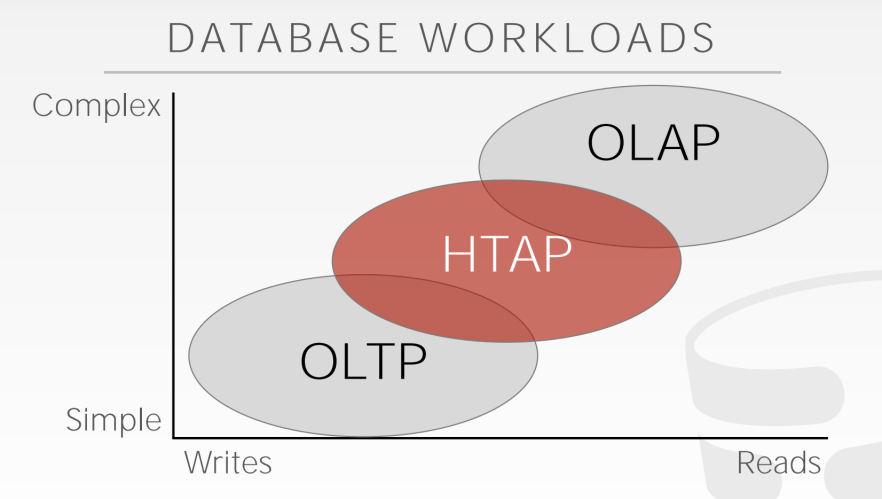
OLAP: Online Analytical Processing

An OLAP workload is characterized by long running, complex queries, reads on large portions of the database. In OLAP worklaods, the database system is analyzing and deriving new data from existing data collected on the OLTP side.

An example of an OLAP workload would be Amazon computing the five most bought items over a one month period for these geographical locations.

HTAP: Hybrid Transaction + Analytical Processing

A new type of workload which has become popular recently is HTAP, which is like a combination which tries to do OLTP and OLAP together on the same database.



3 Storage Models

There are different ways to store tuples in pages. We have assumed the **n-ary storage model** so far.

N-Ary Storage Model (NSM)

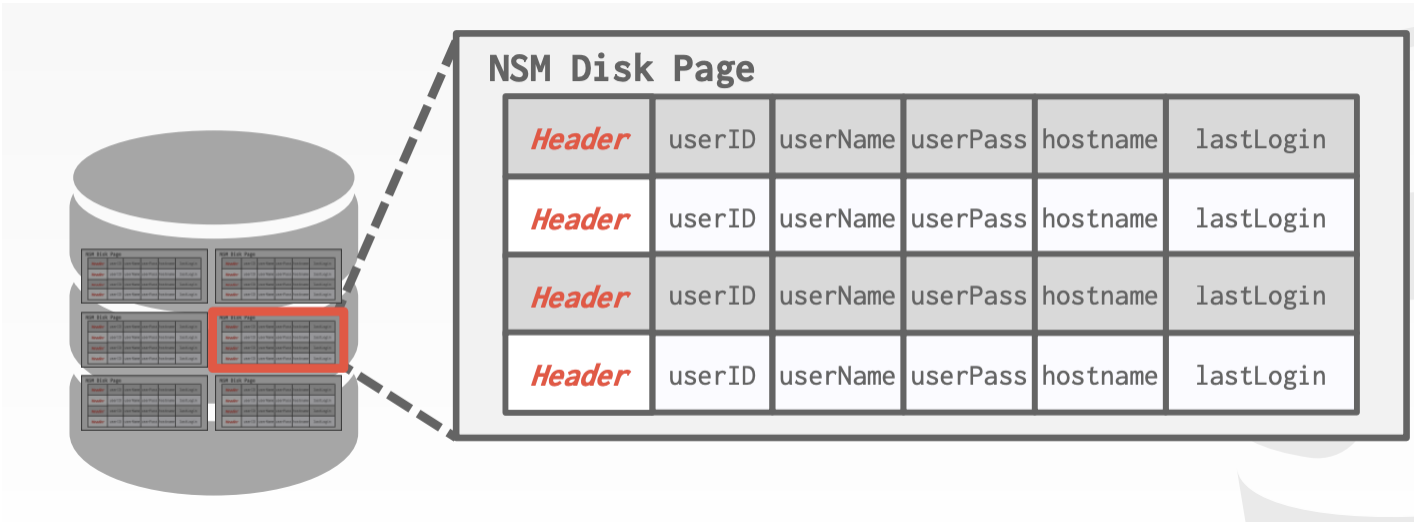
In the n-ary storage model, the DBMS stores all of the attributes for a single tuple contiguously in a single page, so NSM is also known as a "row store." This approach is ideal for OLTP workloads where requests are insert-heavy and transactions tend to operate only an individual entity. It is ideal because it takes only one fetch to be able to get all of the attributes for a single tuple.

Advantages:

- Fast inserts, updates, and deletes.
- Good for queries that need the entire tuple.

Disadvantages:

- Not good for scanning large portions of the table and/or a subset of the attributes. This is because it pollutes the buffer pool by fetching data that is not needed for processing the query.



Decomposition Storage Model (DSM)

In the decomposition storage model, the DBMS stores a single attribute (column) for all tuples contiguously in a block of data. Thus, it is also known as a "column store." This model is ideal for OLAP workloads with many read-only queries that perform large scans over a subset of the table's attributes.

Advantages:

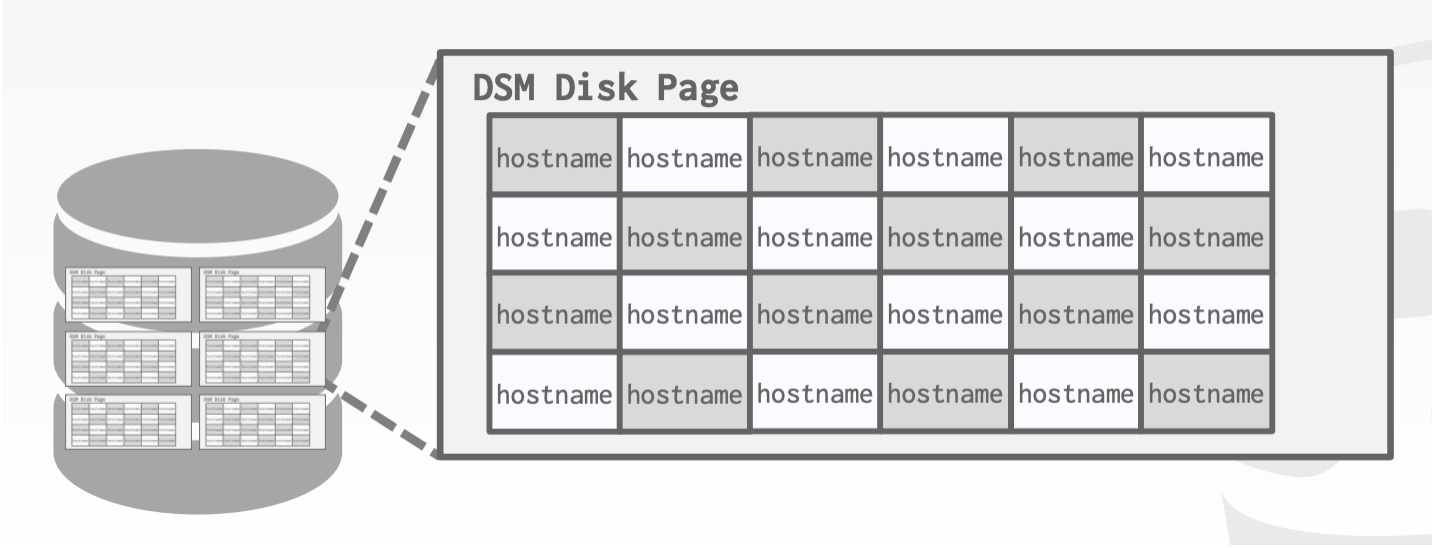
- Reduces the amount of wasted work during query execution because the DBMS only reads the data that it needs for that query.
- Enables better compression because all of the values for the same attribute are stored contiguously.

Disadvantages:

- Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.

To put the tuples back together when using a column store, there are two common approaches: The most commonly used approach is *fixed-length offsets*. Assuming the attributes are all fixed-length, the DBMS can compute the offset of the attribute for each tuple. Then when the system wants the attribute for a specific tuple, it knows how to jump to that spot in the file from the offest. To accommodate the variable-length fields, the system can either pad fields so that they are all the same length or use a dictionary that takes a fixed-size integer and maps the integer to the value.

A less common approach is to use *embedded tuple ids*. Here, for every attribute in the columns, the DBMS stores a tuple id (ex: a primary key) with it. The system then would also store a mapping to tell it how to jump to every attribute that has that id. Note that this method has a large storage overhead because it needs to store a tuple id for every attribute entry.



Lecture #05: Buffer Pools

15-445/645 Database Systems (Fall 2021)

https://15445.courses.cs.cmu.edu/fall2021/

Carnegie Mellon University
Andrew Crotty

1 Introduction

The DBMS is responsible for managing its memory and moving data back-and-forth from the disk. Since, for the most part, data cannot be directly operated on in the disk, any database must be able to efficiently move data represented as files on its disk into memory so that it can be used. A diagram of this interaction is shown in Figure 1. A obstacle that DBMS's face is the problem of minimizing the slowdown of moving data around. Ideally, it should "appear" as if the data is all in the memory already.

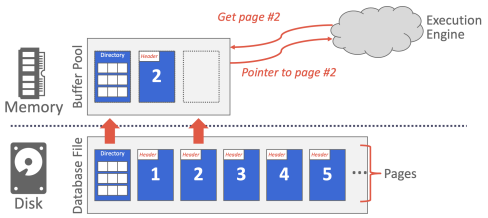


Figure 1: Disk-oriented DBMS.

Another way to think of this problem is in terms of spatial and temporal control.

Spatial Control refers to where pages are physically written on disk. The goal of spatial control is to keep pages that are used together often as physically close together as possible on disk.

Temporal Control refers to when to read pages into memory and when to write them to disk. Temporal control aims to minimize the number of stalls from having to read data from disk.

2 Locks vs. Latches

We need to make a distinction between locks and latches when discussing how the DBMS protects its internal elements.

Locks: A lock is a higher-level, logical primitive that protects the contents of a database (e.g., tuples, tables, databases) from other transactions. Transactions will hold a lock for its entire duration. Database systems can expose to the user which locks are being held as queries are run. Locks need to be able to rollback changes.

Latches: A latch is a low-level protection primitive that the DBMS uses for the critical sections in its internal data structures (e.g., hash tables, regions of memory). Latches are held for only the duration of the operation being made. Latches do not need to be able to rollback changes.

3 Buffer Pool

The *buffer pool* is an in-memory cache of pages read from disk. It is essentially a large memory region allocated inside of the database to store pages that are fetched from disk.

The buffer pool's region of memory organized as an array of fixed size pages. Each array entry is called a *frame*. When the DBMS requests a page, an exact copy is placed into one of the frames of the buffer pool. Then, the database system can search the buffer pool first when a page is requested. If the page is not found, then the system fetches a copy of the page from the disk. See Figure 2 for a diagram of the buffer pool's memory organization.

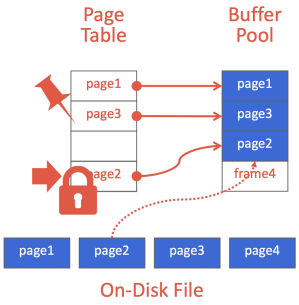


Figure 2: Buffer pool organization and meta-data

Buffer Pool Meta-data

The buffer pool must maintain certain meta-data in order to be used efficiently and correctly.

Firstly, the *page table* is an in-memory hash table that keeps track of pages that are currently in memory. It maps page ids to frame locations in the buffer pool. Since the order of pages in the buffer pool does not necessarily reflect the order on the disk, this extra indirection layer allows for the identification of page locations in the pool. Note that the page table is not to be confused with the *page directory*, which is the mapping from page ids to page locations in database files

The page table also maintains additional meta-data per page, a dirty-flag and a pin/reference counter.

The *dirty-flag* is set by a thread whenever it modifies a page. This indicates to storage manager that the page must be written back to disk.

The *pin/reference Counter* tracks the number of threads that are currently accessing that page (either reading or modifying it). A thread has to increment the counter before they access the page. If a page's count is greater than zero, then the storage manager is not allowed to evict that page from memory.

Memory Allocation Policies

Memory in the database is allocated for the buffer pool according to two policies.

Global policies deal with decisions that the DBMS should make to benefit the entire workload that is being executed. It considers all active transactions to find an optimal decision for allocating memory.

An alternative is *local policies* makes decisions that will make a single query or transaction run faster, even if it isn't good for the entire workload. Local policies allocate frames to a specific transactions without considering the behavior of concurrent transactions.

Most systems use a combination of both global and local views.

4 Buffer Pool Optimizations

There are a number of ways to optimize a buffer pool to tailor it to the application's workload.

Multiple Buffer Pools

The DBMS can maintain multiple buffer pools for different purposes (i.e per-database buffer pool, per-page type buffer pool). Then, each buffer pool can adopt local policies tailored for the data stored inside of it. This method can help reduce latch contention and improves locality.

Two approaches to mapping desired pages to a buffer pool are object IDs and hashing.

Object IDs involve extending the record IDs to include meta-data about what database objects each buffer pool is managing. Then through the object identifier, a mapping from objects to specific buffer pools can be maintained.

Another approach is *hashing* where the DBMS hashes the page id to select which buffer pool to access.

Pre-fetching

The DBMS can also optimize by pre-fetching pages based on the query plan. Then, while the first set of pages is being processed, the second can be pre-fetched into the buffer pool. This method is commonly used by DBMS's when accessing many pages sequentially.

Scan Sharing

Query cursors can reuse data retrieved from storage or operator computations. This allows multiple queries to attach to a single cursor that scans a table. If a query starts a scan and if there one already doing this, then the DBMS will attach to the second query's cursor. The DBMS keeps track of where the second query joined with the first so that it can finish the scan when it reaches the end of the data structure.

Buffer Pool Bypass

The sequential scan operator will not store fetched pages in the buffer pool to avoid overhead. Instead, memory is local to the running query. This works well if operator needs to read a large sequence of pages that are contiguous on disk. Buffer Pool Bypass can also be used for temporary data (sorting, joins).

5 OS Page Cache

Most disk operations go through the OS API. Unless explicitly told otherwise, the OS maintains its own filesystem cache.

Most DBMS use direct I/O to bypass the OS's cache in order to avoid redundant copies of pages and having to manage different eviction policies

Postgres is an example of a database system that uses the OS's Page Cache.

6 Buffer Replacement Policies

When the DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool.

A replacement policy is an algorithm that the DBMS implements that makes a decision on which pages to evict from buffer pool when it needs space.

Implementation goals of replacement policies are improved correctness, accuracy, speed, and meta-data overhead.

Least Recently Used (LRU)

The Least Recently Used replacement policy maintains a timestamp of when each page was last accessed. This timestamp can be stored in a separate data structure, such as a queue, to allow for sorting and improve efficiency. The DBMS picks to evict the page with the oldest timestamp. Additionally, pages are kept in sorted order to reduce sort time on eviction

CLOCK

The CLOCK policy is an approximation of LRU without needing a separate timestamp per page. In the CLOCK policy, each page is given a reference bit. When a page is accessed, set to 1.

To visualize this, organize the pages in a circular buffer with a "clock hand". Upon sweeping check if a page's bit is set to 1. If yes, set to zero, if no, then evict it. In this way, the clock hand remembers position between evictions.

Alternatives

There are a number of problems with LRU and CLOCK replacement policies.

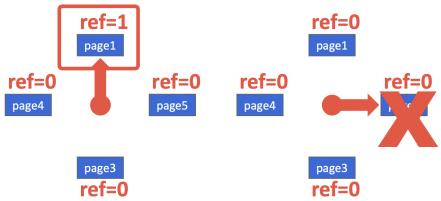


Figure 3: Visualization of CLOCK replacement policy. Page 1 is referenced and set to 1. When the clock hand sweeps, it sets the reference bit for page 1 to 0 and evicts page 5.

Namely, LRU and CLOCK are susceptible to *sequential flooding*, where the buffer pool's contents are corrupted due to a sequential scan. Since sequential scans read every page, the timestamps of pages read may not reflect which pages we actually want. In other words, the most recently used page is actually the most unneeded page.

There are three solutions to address the shortcomings of LRU and CLOCK policies.

One solution is *LRU-K* which tracks the history of the last K references as timestamps and computes the interval between subsequent accesses. This history is used to predict the next time a page is going to be accessed.

Another optimization is *localization* per query. The DBMS chooses which pages to evict on a per transaction/query basis. This minimizes the pollution of the buffer pool from each query.

Lastly, *priority hints* allow transactions to tell the buffer pool whether page is important or not based on the context of each page during query execution.

Dirty Pages

There are two methods to handling pages with dirty bits. The fastest option is to drop any page in the buffer pool that is not dirty. A slower method is to write back dirty pages to disk to ensure that its changes are persisted.

These two methods illustrate the trade-off between fast evictions versus dirty writing pages that will not be read again in the future.

One way to avoid the problem of having to write out pages unnecessarily is *background writing*. Through background writing, the DBMS can periodically walk through the page table and write dirty pages to disk. When a dirty page is safely written, the DBMS can either evict the page or just unset the dirty flag.

7 Other Memory Pools

The DBMS needs memory for things other than just tuples and indexes. These other memory pools may not always backed by disk depending on implementation.

- Sorting + Join Buffers
- Query Caches
- Maintenance Buffers
- Log Buffers
- Dictionary Caches

Lecture #06: Hash Tables

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Andrew Crotty

1 Data Structures

A DBMS uses various data structures for many different parts of the system internals. Some examples include:

- **Internal Meta-Data:** This is data that keeps track of information about the database and the system state.
  Ex: Page tables, page directories
- **Core Data Storage:** Data structures are used as the base storage for tuples in the database.
- **Temporary Data Structures:** The DBMS can build data structures on the fly while processing a query to speed up execution (e.g., hash tables for joins).
- **Table Indices:** Auxiliary data structures can be used to make it easier to find specific tuples.

There are two main design decisions to consider when implementing data structures for the DBMS:

1. Data organization: We need to figure out how to layout memory and what information to store inside the data structure in order to support efficient access.
2. Concurrency: We also need to think about how to enable multiple threads to access the data structure without causing problems.

2 Hash Table

A hash table implements an associative array abstract data type that maps keys to values. It provides on average $O(1)$ operation complexity ($O(n)$ in the worst-case) and $O(n)$ storage complexity. Note that even with $O(1)$ operation complexity on average, there are constant factor optimizations which are important to consider in the real world.

A hash table implementation is comprised of two parts:

- **Hash Function:** This tells us how to map a large key space into a smaller domain. It is used to compute an index into an array of buckets or slots. We need to consider the trade-off between fast execution and collision rate. On one extreme, we have a hash function that always returns a constant (very fast, but everything is a collision). On the other extreme, we have a "perfect" hashing function where there are no collisions, but would take extremely long to compute. The ideal design is somewhere in the middle.
- **Hashing Scheme:** This tells us how to handle key collisions after hashing. Here, we need to consider the trade-off between allocating a large hash table to reduce collisions and having to execute additional instructions when a collision occurs.

3 Hash Functions

A *hash function* takes in any key as its input. It then returns an integer representation of that key (i.e., the "hash"). The function's output is deterministic (i.e., the same key should always generate the same hash output).

The DBMS need not use a cryptographically secure hash function (e.g., SHA-256) because we do not need to worry about protecting the contents of keys. These hash functions are primarily used internally by the DBMS and thus information is not leaked outside of the system. In general, we only care about the hash function's speed and collision rate.

The current state-of-the-art hash function is Facebook XXHash3.

4 Static Hashing Schemes

A static hashing scheme is one where the size of the hash table is fixed. This means that if the DBMS runs out of storage space in the hash table, then it has to rebuild a larger hash table from scratch, which is very expensive. Typically the new hash table is twice the size of the original hash table.

To reduce the number of wasteful comparisons, it is important to avoid collisions of hashed key. Typically, we use twice the number of slots as the number of expected elements.

The following assumptions usually do not hold in reality:

1. The number of elements is known ahead of time.
2. Keys are unique.
3. There exists a perfect hash function.

Therefore, we need to choose the hash function and hashing schema appropriately.

4.1 Linear Probe Hashing

This is the most basic hashing scheme. It is also typically the fastest. It uses a circular buffer of array slots. The hash function maps keys to slots. When a collision occurs, we linearly seach the adjacent slots until an open one is found. For lookups, we can check the slot the key hashes to, and search linearly until we find the desired entry (or an empty slot, in which case the key is not in the table). Note that this means we have to store the key in the slot as well so that we can check if an entry is the desired one. Deletions are more tricky. We have to be careful about just removing the entry from the slot, as this may prevent future lookups from finding entries that have been put below the now empty slot. There are two solutions to this problem:

- The most common approach is to use "tombstones". Instead of deleting the entry, we replace it with a "tombstone" entry which tells future lookups to keep scanning.
- The other option is to shift the adjacent data after deleting an entry to fill the now empty slot. However, we must be careful to only move the entries which were originally shifted.

Non-unique Keys: In the case where the same key may be associated with multiple different values or tuples, there are two approaches.

- Seperate Linked List: Instead of storing the values with the keys, we store a pointer to a seperate storage area which contains a linked list of all the values.
- Redundant Keys: The more common approach is to simply store the same key multiple times in the table. Everything with linear probing still works even if we do this.

4.2 Robin Hood Hashing

This is an extension of linear probe hashing that seeks to reduce the maximum distance of each key from their optimal position (i.e. the original slot they were hashed to) in the hash table. This strategy steals slots from "rich" keys and gives them to "poor" keys.

In this variant, each entry also records the "distance" they are from their optimal position. Then, on each insert, if the key being inserted would be farther away from their optimal position at the current slot than the current entry's distance, we replace the current entry and continue trying to insert the old entry farther down the table.

4.3 Cuckoo Hashing

Instead of using a single hash table, this approach maintains multiple has tables with different hash functions. The hash functions are the same algorithm (e.g., XXHash, CityHash); they generate different hashes for the same key by using different seed values.

When we insert, we check every table and choose one that has a free slot (if multiple have one, we can compare things like load factor, or more commonly, just choose a random table). If no table has a free slot, we choose (typically a random one) and evict the old entry. We then rehash the old entry into a different table. In rare cases, we may end up in a cycle. If this happens, we can rebuild all of the hash tables with new hash function seeds (less common) or rebuild the hash tables using larger tables (more common).

Cuckoo hashing guarantees $O(1)$ lookups and deletions, but insertions may be more expensive.

5 Dynamic Hashing Schemes

The static hashing schemes require the DBMS to know the number of elements it wants to store. Otherwise it has to rebuild the table if it needs to grow/shrink in size.

Dynamic hashing schemes are able to resize the hash table on demand without needing to rebuild the entire table. The schemes perform this resizing in different ways that can either maximize reads or writes.

5.1 Chained Hashing

This is the most common dynamic hashing scheme. The DBMS maintains a linked list of buckets for each slot in the hash table. Keys which hash to the same slot are simply inserted into the linked list for that slot.

5.2 Extendible Hashing

Improved variant of chained hashing that splits buckets instead of letting chains to grow forever. This approach allows multiple slot locations in the hash table to point to the same bucket chain.

The core idea behind re-balancing the hash table is to to move bucket entries on split and increase the number of bits to examine to find entries in the hash table. This means that the DBMS only has to move data within the buckets of the split chain; all other buckets are left untouched.

- The DBMS maintains a global and local depth bit counts that determine the number bits needed to find buckets in the slot array.
- When a bucket is full, the DBMS splits the bucket and reshuffle its elements. If the local depth of the split bucket is less than the global depth, then the new bucket is just added to the existing slot array. Otherwise, the DBMS doubles the size of the slot array to accommodate the new bucket and increments the global depth counter.

5.3 Linear Hashing

Instead of immediately splitting a bucket when it overflows, this scheme maintains a *split pointer* that keeps track of the next bucket to split. No matter whether this pointer is pointing to a bucket that overflowed, the DBMS always splits. The overflow criterion is left up to the implementation.

- When any bucket overflows, split the bucket at the pointer location by adding a new slot entry, and create a new hash function.
- If the hash function maps to slot that has previously been pointed to by pointer, apply the new hash function.
- When the pointer reaches last slot, delete original hash function and replace it with a new hash function.

Lecture #07: Tree Indexes

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Andrew Crotty

1 Table Indexes

There are a number of different data structures one can use inside of a database system for purposes such as internal meta-data, core data storage, temporary data structures, or table indexes. For table indexes, which may involve queries with range scans,

A *table index* is a replica of a subset of a table's columns that is organized and/or sorted for efficient access using a subset of those attributes. So instead of performing a sequential scan, the DBMS can lookup the table index's auxiliary data structure to find tuples more quickly. The DBMS ensures that the contents of the tables and the indexes are always logically in sync.

There exists a trade-off between the number of indexes to create per database. Although more indexes makes looking up queries faster, indexes also use storage and require maintenance. It is the DBMS's job to figure out the best indexes to use to execute queries.

2 B+Tree

A *B+Tree* is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertion, and deletions in $O(log(n))$. It is optimized for disk-oriented DBMS's that read/write large blocks of data.

Almost every modern DBMS that supports order-preserving indexes uses a B+Tree. There is a specific data structure called a *B-Tree*, but people also use the term to generally refer to a class of data structures. The primary difference between the original *B-Tree* and the B+Tree is that B-Trees stores keys and values in *all nodes*, while B+ trees store values *only in leaf nodes*. Modern B+Tree implementations combine features from other B-Tree variants, such as the sibling pointers used in the B^{link}-Tree.

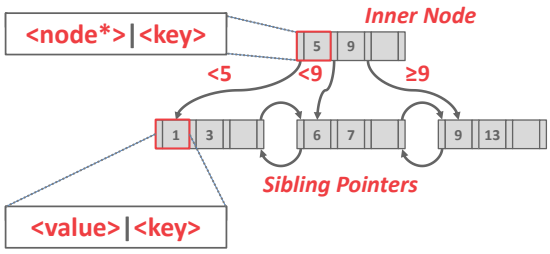


Figure 1: B+ Tree diagram

Formally, a B+Tree is an M-way search tree with the following properties:

- It is perfectly balanced (i.e., every leaf node is at the same depth).
- Every inner node other than the root is at least half full ($M/2 - 1 <=$ num of keys $<= M - 1$).
- Every inner node with k keys has k+1 non-null children.

Every node in a B+Tree contains an array of key/value pairs. The keys in these pairs are derived from the attribute(s) that the index is based on. The values will differ based on whether a node is an inner node or a leaf node. For inner nodes, the value array will contain pointers to other nodes. Two approaches for leaf node values are *record IDs* and *tuple data*. Record IDs refer to a pointer to the location of the tuple. Leaf nodes that have tuple data store the the actual contents of the tuple in each node.

Though it is not necessary according to the definition of the B+Tree, arrays at every node are almost always sorted by the keys.

Selection Conditions

Because B+Trees are in sorted order, look ups have fast traversal and also do not require the entire key. The DBMS can use a B+Tree index if the query provides any of the attributes of the search key. This differs from a hash index, which requires all attributes in the search key.

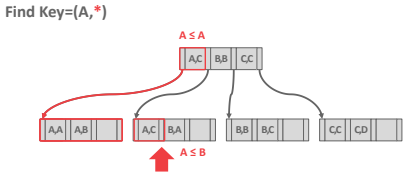


Figure 2: To perform a prefix search on a B+Tree, one looks at the first attribute on the key, follows the path down and performs a sequential scan across the leaves to find all they keys that one wants.

Insertion

To insert a new entry into a B+Tree, one must traverse down the tree and use the inner nodes to figure out which leaf node to insert the key into.

1. Find correct leaf L.
2. Add new entry into L in sorted order:
   - If L has enough space, the operation is done.
   - Otherwise split L into two nodes L and L_2. Redistribute entries evenly and copy up middle key. Insert index entry pointing to L_2 into parent of L.
3. To split an inner node, redistribute entries evenly, but push up the middle key.

Deletion

Whereas in inserts we occasionally had to split leaves when the tree got too full, if a deletion causes a tree to be less than half-full, we must merge in order to re-balance the tree.

1. Find correct leaf L.
2. Remove the entry:
   - If L is at least half full, the operation is done.
   - Otherwise, you can try to redistribute, borrowing from sibling.

- If redistribution fails, merge L and sibling.

3. If merge occurred, you must delete entry in parent pointing to L.

Non-Unique Indexes

Like in hash tables, B+Trees can deal with non-unique indexes by duplicating keys or storing value lists. In the duplicate keys approach, the same leaf node layout is used but duplicate keys are stored multiple times. In the value lists approach, each key is stored only once and maintains a linked list of unique values.

Duplicate Keys

There are two approaches to duplicate keys in a B+Tree.

The first approach is to *append record ID*s as part of the key. Since each tuple's record ID is unique, this will ensure that all the keys are identifiable.

The second approach is to allow leaf nodes to spill into *overflow nodes* that contain the duplicate keys. Although no redundant information is stored, this approach is more complex to maintain and modify.

Clustered Indexes

The table is stored in the sort order specified by the primary key, as either heap- or index-organized storage. Since some DBMSs always use a clustered index, they will automatically make a hidden row id primary key if a table doesn't have an explicit one, but others cannot use them at all.

Heap Clustering

Tuples are sorted in the heap's pages using the order specified by a clustering index. DBMS can jump directly to the pages if clustering index's attributes are used to access tuples.

Index Scan Page Sorting

Since directly retrieving tuples from an unclustered index is inefficient, the DBMS can first figure out all the tuples that it needs and then sort them based on their page id.

3 B+Tree Design Choices

3.1 Node Size

Depending on the storage medium, we may prefer larger or smaller node sizes. For example, nodes stored on hard drives are usually on the order of megabytes in size to reduce the number of seeks needed to find data and amortize the expensive disk read over a large chunk of data, while in-memory databases may use page sizes as small as 512 bytes in order to fit the entire page into the CPU cache as well as to decrease data fragmentation. This choice can also be dependent on the type of workload, as point queries would prefer as small a page as possible to reduce the amount of unnecessary extra info loaded, while a large sequential scan might prefer large pages to reduce the number of fetches it needs to do.

3.2 Merge Threshold

While B+Trees have a rule about merging underflowed nodes after a delete, sometimes it may be beneficial to temporarily violate the rule to reduce the number of deletion operation. For instance, eager merging could lead to thrashing, where a lot of successive delete and insert operations lead to constant splits and merges. It also allows for batched merging where multiple merge operations happen all at once, reducing the amount

of time that expensive write latches have to be taken on the tree.

3.3 Variable Length Keys

Currently we have only discussed B+Trees with fixed length keys. However we may also want to support variable length keys, such as the case where a small subset of large keys lead to a lot of wasted space. There are several approaches to this:

1. **Pointers**
   Instead of storing the keys directly, we could just store a pointer to the key. Due to the inefficiency of having to chase a pointer for each key, the only place that uses this method in production is embedded devices, where its tiny registers and cache may benefit from such space savings
2. **Variable Length Nodes**
   We could also still store the keys like normal and allow for variable length nodes. This is infeasible and largely not used due to the significant memory management overhead of dealing with variable length nodes.
3. **Padding**
   Instead of varying the key size, we could set each key's size to the size of the maximum key and pad out all the shorter keys. In most cases this is a massive waste of memory, so you don't see this used by anyone either.
4. **Key Map/Indirection**
   The method that nearly everyone uses is replacing the keys with an index to the key-value pair in a separate dictionary. This offers significant space savings and potentially shortcuts point queries (since the key-value pair the index points to is the exact same as the one pointed to by leaf nodes). Due to the small size of the dictionary index value, there is enough space to place a prefix of each key alongside the index, potentially allowing some index searching and leaf scanning to not even have to chase the pointer (if the prefix is at all different from the search key).

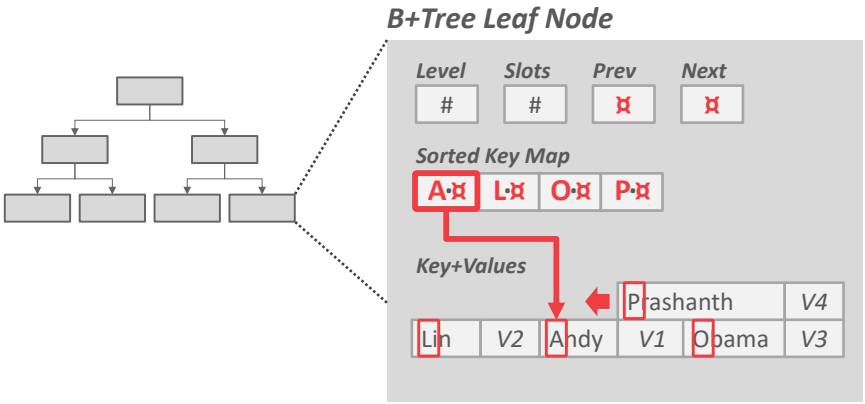


Figure 3: An example of Key Map/Indirection. The map stores a small prefix of the key, as well as a pointer to the key-value pair.

3.4 Intra-Node Search

Once we reach a node, we still need to search within the node (either to find the next node from an inner node, or to find our key value in a leaf node). While this is relatively simple, there are still some tradeoffs to consider:

1. Linear
   The simplest solution is to just scan every key in the node until we find our key. On the one hand, we don't have to worry about sorting the keys, making insertions and deletes much quicker. On the other hand, this is relatively inefficient and has a complexity of $\mathbb{O}(n)$ per search.
2. Binary
   A more efficient solution for searching would be to keep each node sorted and use binary search to find the key. This is as simple as jumping to the middle of a node and pivoting left or right depending on the comparison between the keys. Searches are much more efficient this way, as this method only has the complexity of $\mathbb{O}(\ln(n))$ per search. However, insertions become more expensive as we must maintain the sort of each node.
3. Interpolation
   Finally, in some circumstances we may be able to utilize interpolation to find the key. This method takes advantage of any metadata stored about the node (such as max element, min element, average, etc.) and uses it to generate an approximate location of the key. For example, if we are looking for 8 in a node and we know that 10 is the max key and $10 - (n + 1)$ is the smallest key (where n is the number of keys in each node), then we know to start searching 2 slots down from the max key, as the key one slot away from the max key must be 9 in this case. Despite being the fastest method we have given, this method is only seen in academic databases due to its limited applicability to keys with certain properties (like integers) and complexity.

4 Optimizations

4.1 Prefix Compression

Most of the time when we have keys in the same node there will be some partial overlap of some prefix of each key (as similar keys will end up right next to each other in a sorted B+Tree). Instead of storing this prefix as part of each key multiple times, we can simply store the prefix once at the beginning of the node and then only include the unique sections of each key in each slot.

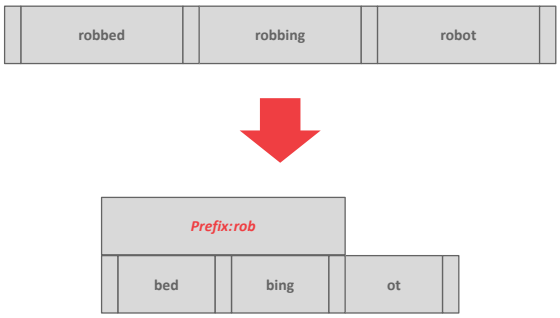


Figure 4: An example of prefix compression. Since the keys are in lexicographic order, they are likely to share some prefix.

4.2 Deduplication

In the case of an index which allows non-unique keys, we may end up with leaf nodes containing the same key over and over with different values attached. One optimization of this could be only writing the key once and then following it with all of its associated values.

4.3 Bulk Insert

When a B+Tree is initially built, having to insert each key the usual way would lead to constant split operations. Since we already give leaves sibling pointers, initial insertion of data is much more efficient if we construct a sorted linked list of leaf nodes and then easily build the index from the bottom up using the first key from each leaf node. Note that depending on our context we may wish to pack the leaves as tightly as possible to save space or leave space in each leaf to allow for more inserts before a split is necessary.

Lecture #08: Index Concurrency Control

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Andrew Crotty

1 Index Concurrency Control

A *concurrency control* protocol is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object.

A protocol's correctness criteria can vary:

- **Logical Correctness:** This means that the thread is able to read values that it should expects to read, e.g. a thread should read back the value it had written previously.
- **Physical Correctness:** This means that the internal representation of the object is sound, e.g. there are not pointers in the data structure that will cause a thread to read invalid memory locations.

The logical contents of the index is the only thing we care about in this lecture. They are not quite like other database elements so we can treat them differently.

2 Locks vs. Latches

There is an important distinction between locks and latches when discussing how the DBMS protects its internal elements.

Locks

A lock is a higher-level, logical primitive that protects the contents of a database (e.g., tuples, tables, databases) from other transactions. Transactions will hold a lock for its entire duration. Database systems can expose to the user the locks that are being held as queries are run. Locks need to be able to rollback changes.

Latches

Latches are the low-level protection primitives used for critical sections the DBMS's internal data structures (e.g., data structure, regions of memory) from other threads. Latches are held for only the duration of the operation being made. Latches do not need to be able to rollback changes. There are two modes for latches:

- **`READ`:** Multiple threads are allowed to read the same item at the same time. A thread can acquire the read latch even if another thread has acquired it as well.
- **`WRITE`:** Only one thread is allowed to access the item. A thread cannot acquire a write latch if another thread holds the latch in any mode. A thread holding a write latch also prevents other threads from acquiring a read latch.

3 Latch Implementations

The underlying primitive that used to implement a latch is through an atomic *compare-and-swap* (CAS) instruction that modern CPUs provide. With this, a thread can check the contents of a memory location to

see whether it has a certain value. If it does, then the CPU will swap the old value with a new one. Otherwise the memory location remains unmodified.

There are several approaches to implementing a latch in a DBMS. Each approach has different trade-offs in terms of engineering complexity and runtime performance. These test-and-set steps are performed atomically (i.e., no other thread can update the value in between the test and set steps.

Blocking OS Mutex

One possible implementation of latches is the OS built-in mutex infrastructure. Linux provides the futex (fast user-space mutex), which is comprised of (1) a spin latch in user-space and (2) an OS-level mutex. If the DBMS can acquire the user-space latch, then the latch is set. It appears as a single latch to the DBMS even though it contains two internal latches. If the DBMS fails to acquire the user-space latch, then it goes down into the kernel and tries to acquire a more expensive mutex. If the DBMS fails to acquire this second mutex, then the thread notifies the OS that it is blocked on the mutex and then it is descheduled.

OS mutex is generally a bad idea inside of DBMSs as it is managed by OS and has large overhead.

- **Example:** `std::mutex`
- **Advantages:** Simple to use and requires no additional coding in DBMS.
- **Disadvantages:** Expensive and non-scalable (about 25 ns per lock/unlock invocation) because of OS scheduling.

Test-and-Set Spin Latch (TAS)

Spin latches are a more efficient alternative to an OS mutex as it is controlled by the DBMSs. A spin latch is essentially a location in memory that threads try to update (e.g., setting a boolean value to true). A thread performs CAS to attempt to update the memory location. The DBMS can control what happens if it fails to get the latch. It can choose to try again (for example, using a while loop) or allow the OS to deschedule it. Thus, this method gives the DBMS more control than the OS mutex, where failing to acquire a latch gives control to the OS.

- **Example:** `std::atomic<T>`
- **Advantages:** Latch/unlatch operations are efficient (single instruction to lock/unlock).
- **Disadvantages:** Not scalable nor cache-friendly because with multiple threads, the CAS instructions will be executed multiple times in different threads. These wasted instructions will pile up in high contention environments; the threads look busy to the OS even though they are not doing useful work. This leads to cache coherence problems because threads are polling cache lines on other CPUs.

Reader-Writer Latches

Mutexes and Spin Latches do not differentiate between reads/writes (i.e., they do not support different modes). The DBMS needs a way to allow for concurrent reads, so if the application has heavy reads it will have better performance because readers can share resources instead of waiting.

A Reader-Writer Latch allows a latch to be held in either read or write mode. It keeps track of how many threads hold the latch and are waiting to acquire the latch in each mode. Reader-writer latches use one of the previous two latch implementations as primitives and have additional logic to handle reader-writer queues, which are queues requests for the latch in each mode. Different DBMSs can have different policies for how it handles the queues.

- **Advantages:** Allows for concurrent readers.

- **Disadvantages:** The DBMS has to manage read/write queues to avoid starvation. Larger storage overhead than Spin Latches due to additional meta-data.

4 Hash Table Latching

It is easy to support concurrent access in a static hash table due to the limited ways threads access the data structure. For example, all threads move in the same direction when moving from slot to the next (i.e., top-down). Threads also only access a single page/slot at a time. Thus, deadlocks are not possible in this situation because no two threads could be competing for latches held by the other. When we need to resize the table, we can just take a global latch on the entire table to perform the operation.

Latching in a dynamic hashing scheme (e.g., extendible) is a more complicated scheme because there is more shared state to update, but the general approach is the same.

There are two approaches to support latching in a hash table that differ on the granularity of the latches:

- **Page Latches:** Each page has its own Reader-Writer latch that protects its entire contents. Threads acquire either a read or write latch before they access a page. This decreases parallelism because potentially only one thread can access a page at a time, but accessing multiple slots in a page will be fast for a single thread because it only has to acquire a single latch.
- **Slot Latches:** Each slot has its own latch. This increases parallelism because two threads can access different slots on the same page. But it increases the storage and computational overhead of accessing the table because threads have to acquire a latch for every slot they access, and each slot has to store data for the latches. The DBMS can use a single mode latch (i.e., Spin Latch) to reduce meta-data and computational overhead at the cost of some parallelism.

It is also possible to create a latch-free linear probing hash table directly using compare-and-swap (CAS) instructions. Insertion at a slot can be achieved by attempting to compare-and-swap a special ”null” value with the tuple we wish to insert. If this fails, we can probe the next slot, continuing until it succeeds.

5 B+Tree Latching

The challenge of B+Tree latching is preventing the two following problems:

- Threads trying to modify the contents of a node at the same time.
- One thread traversing the tree while another thread splits/merges nodes.

Latch crabbing/coupling is a protocol to allow multiple threads to access/modify B+Tree at the same time. The basic idea is as follows.

1. Get latch for the parent.
2. Get latch for the child.
3. Release latch for the parent if it is deemed “safe”. A “safe” node is one that will not split (not full on insertion) or merge when updated (more than half full on deletion).

Basic Latch Crabbing Protocol:

- **Search:** Start at the root and go down, repeatedly acquire latch on the child and then unlatch parent.
- **Insert/Delete:** Start at the root and go down, obtaining `X` latches as needed. Once the child is latched, check if it is safe. If the child is safe, release latches on all its ancestors.

Note that read latches do not need to worry about the “safe” condition. The notion of “safe” also depends on whether the operation is an insertion or a deletion. A full node is “safe” for deletion since a merge will not be needed but is not “safe” for an insertion since we may need to split the node.

The order in which latches are released is not important from a correctness perspective. However, from a performance point of view, it is better to release the latches that are higher up in the tree since they block access to a larger portion of leaf nodes.

Improved Latch Crabbing Protocol: The problem with the basic latch crabbing algorithm is that transactions **always** acquire an exclusive latch on the root for every insert/delete operation. This limits parallelism. Instead, one can assume that having to resize (i.e., split/merge nodes) is rare, and thus transactions can acquire shared latches down to the leaf nodes. Each transaction will assume that the path to the target leaf node is safe, and use `READ` latches and crabbing to reach it and verify. If the leaf node is not safe, then we abort and do the previous algorithm where we acquire `WRITE` latches.

- **Search:** Same algorithm as before.
- **Insert/Delete:** Set `READ` latches as if for search, go to leaf, and set `WRITE` latch on leaf. If the leaf is not safe, release all previous latches, and restart the transaction using previous Insert/Delete protocol.

Leaf Node Scans

The threads in these protocols acquire latches in a "top-down" manner. This means that a thread can only acquire a latch from a node that is below its current node. If the desired latch is unavailable, the thread must wait until it becomes available. Given this, there can never be deadlocks.

Leaf node scans are susceptible to deadlocks because now we have threads trying to acquire locks in two different directions at the same time (i.e., left-to-right and right-to-left). Index latches do not support deadlock detection or avoidance.

Thus, the only way programmers can deal with this problem is through coding discipline. The leaf node sibling latch acquisition protocol must support a "no-wait" mode. That is, the B+tree code must cope with failed latch acquisitions. This means that if a thread tries to acquire a latch on a leaf node but that latch is unavailable, then it will immediately abort its operation (releasing any latches that it holds) and then restart the operation.

Lecture #9: Sorting & Aggregation Algorithms

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Andrew Crotty

1 Sorting

DBMSs need to sort data because tuples in a table have no specific order under the relation model. Sorting is (potentially) used in `ORDER BY`, `GROUP BY`, `JOIN`, and `DISTINCT` operators. If the data that that needs to be sorted fits in memory, then the DBMS can use a standard sorting algorithms (e.g., quicksort). If the data does not fit, then the DBMS needs to use external sorting that is able to spill to disk as needed and prefers sequential over random I/O.

The standard algorithm for sorting data which is too large to fit in memory is **external merge sort**. It is a divide-and-conquer sorting algorithm that splits the data set into separate *runs* and then sorts them individually. It can spill runs to disk as needed then read them back in one at a time. The algorithm is comprised of two phases:

Phase #1 – Sorting: First, the algorithm sorts small chunks of data that fit in main memory, and then writes the sorted pages back to disk.

Phase #2 – Merge: Then, the algorithm combines the sorted sub-files into a larger single file.

Two-way Merge Sort

The most basic version of the algorithm is the two-way merge sort. The algorithm reads each page during the sorting phase, sorts it, and writes the sorted version back to disk. Then, in the merge phase, it uses three buffer pages. It reads two sorted pages in from disk, and merges them together into a third buffer page. Whenever the third page fills up, it is written back to disk and replaced with an empty page. Each set of sorted pages is called a *run*. The algorithm then recursively merges the runs together.

If N is the total number of data pages, the algorithm makes $1 + \lceil log_2 N \rceil$ total passes through the data (1 for the first sorting step then $\lceil log_2 N \rceil$ for the recursive merging). The total I/O cost is $2N \times$ (# of passes) since each pass performs an I/O read and an I/O write for each page.

General (K-way) Merge Sort

The generalized version of the algorithm allows the DBMS to take advantage of using more than three buffer pages. Let B be the total number of buffer pages available. Then, during the sort phase, the algorithm can read B pages at a time and write $\lceil \frac{N}{B} \rceil$ sorted runs back to disk. The merge phase can also combine up to $B - 1$ runs in each pass, again using one buffer page for the combined data and writing back to disk as needed.

In the generalized version, the algorithm performs $1 + \lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil$ passes (one for the sorting phase and $\lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil$ for the merge phase. Then, the total I/O cost is $2N \times$ (# of passes) since it again has to make a read and write for each page in each pass.

Double Buffering Optimization

One optimization for external merge sort is prefetching the next run in the background and storing it in a second buffer while the system is processing the current run. This reduces the wait time for I/O requests at each step by continuously utilizing the disk. This optimization requires the use of multiple threads, since the prefetching should occur while the computation for the current run is happening.

Using B+Trees

It is sometimes advantageous for the DBMS to use an existing B+tree index to aid in sorting rather than using the external merge sort algorithm. In particular, if the index is a clustered index, the DBMS can just traverse the B+tree. Since the index is clustered, the data will be stored in the correct order, so the I/O access will be sequential. This means it is always better than external merge sort since no computation is required. On the other hand, if the index is unclustered, traversing the tree is almost always worse, since each record could be stored in any page, so nearly all record accesses will require a disk read.

2 Aggregations

An aggregation operator in a query plan collapses the values of one or more tuples into a single scalar value. There are two approaches for implementing an aggregation: (1) sorting and (2) hashing.

Sorting

The DBMS first sorts the tuples on the `GROUP BY` key(s). It can use either an in-memory sorting algorithm if everything fits in the buffer pool (e.g., quicksort) or the external merge sort algorithm if the size of the data exceeds memory. The DBMS then performs a sequential scan over the sorted data to compute the aggregation. The output of the operator will be sorted on the keys.

When performing sorting aggregations, it is important to order the query operations to maximize efficiency. For example, if the query requires a filter, it is better to perform the filter first and then sort the filtered data to reduce the amount of data that needs to be sorted.

Hashing

Hashing can be computationally cheaper than sorting for computing aggregations. The DBMS populates an ephemeral hash table as it scans the table. For each record, check whether there is already an entry in the hash table and perform the appropriate modification. If the size of the hash table is too large to fit in memory, then the DBMS has to spill it to disk. There are two phases to accomplishing this:

- **Phase #1 – Partition:** Use a hash function h_1 to split tuples into partitions on disk based on target hash key. This will put all tuples that match into the same partition. The DBMS spills partitions to disk via output buffers.
- **Phase #2 – ReHash:** For each partition on disk, read its pages into memory and build an in-memory hash table based on a second hash function h_2 (where $h_1 \neq h_2$). Then go through each bucket of this hash table to bring together matching tuples to compute the aggregation. This assumes that each partition fits in memory.

During the ReHash phase, the DBMS can store pairs of the form (`GroupByKey`→`RunningValue`) to compute the aggregation. The contents of `RunningValue` depends on the aggregation function. To insert a new tuple into the hash table:

- If it finds a matching `GroupByKey`, then update the `RunningValue` appropriately.
- Else insert a new (`GroupByKey`→`RunningValue`) pair.

Lecture #10: Joins Algorithms

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Andrew Crotty

1 Joins

The goal of a good database design is to minimize the amount of information repetition. This is why tables are composed based on normalization theory. Joins are therefore needed to reconstruct the original tables.

This class will cover **inner equijoin** algorithms for combining two-tables. An *equijoin* algorithm joins tables where keys are equal. These algorithms can be tweaked to support other joins.

Operator Output

For a tuple $r \in R$ and a tuple $s \in S$ that match on join attributes, the join operator concatenates r and s together into a new output tuple.

In reality, contents of output tuples generated by a join operator varies. It depends on the DBMS's query processing model, storage model, and the query itself. There are multiple approaches to the contents of the join operator output.

- **Data:** This approach copies the values for the attributes in the outer and inner tables into tuples put into an intermediate result table just for that operator. The advantage of this approach is that future operators in the query plan never need to go back to the base tables to get more data. The disadvantage is that this requires more memory to materialize the entire tuple. This is called *early materialization*. The DBMS can also do additional computation and omit attributes which will not be needed later in the query to further optimize this approach.
- **Record Ids:** In this approach, the DBMS only copies the join keys along with the record ids of the matching tuples. This approach is ideal for column stores because the DBMS does not copy data that is not needed for the query. This is called *late materialization*.

Cost Analysis

The cost metric used here to analyze the different join algorithms will be the number of disk I/Os used to compute the join. This includes I/Os incurred by reading data from disk as well as writing any intermediate data out to disk. Note that only I/Os from computing the join are considered, while I/O incurred when outputting the result is not. This is because the output for any algorithm will be the same, so the output cost will not change among different algorithms.

Variables used in this lecture:
- M pages in table R (Outer Table), m tuples total
- N pages in table S (Inner Table), n tuples total

In general, there will be many algorithms/optimizations which can reduce join costs in some cases, but no single algorithm which works well in every scenario.

2 Nested Loop Join

At a high-level, this type of join algorithm is comprised of two nested `for` loops that iterate over the tuples in both tables and compares each unique of them. If the tuples match the join predicate, then output them. The table in the outer `for` loop is called the *outer table*, while the table in the inner `for` loop is called the *inner table*.

The DBMS will always want to use the "smaller" table as the outer table. Smaller can be in terms of the number of tuples or number of pages. The DBMS will also want to buffer as much of the outer table in memory as possible. It can also try to leverage an index to find matches in inner table.

Simple Nested Loop Join

For each tuple in the outer table, compare it with each tuple in the inner table. This is the worst case scenario where the DBMS must do an entire scan of the inner table for each tuple in the outer table without any caching or access locality.
Cost: $M + (m \times N)$

Block Nested Loop Join

For each block in the outer table, fetch each block from the inner table and compare all the tuples in those two blocks. This algorithm performs fewer disk access because the DBMS scans the inner table for every outer table block instead of for every tuple.
Cost: $M + (M \times N)$

If the DBMS has B buffers available to compute the join, then it can use $B - 2$ buffers to scan the outer table. It will use one buffer to scan the inner table and one buffer to store the output of the join.
Cost: $M + \left(\left\lceil \frac{M}{B-2} \right\rceil \times N\right)$

Index Nested Loop Join

The previous nested loop join algorithms perform poorly because the DBMS has to do a sequential scan to check for a match in the inner table. However, if the database already has an index for one of the tables on the join key, it can use that to speed up the comparison. The DBMS can either use an existing index or build a temporary one for the join operation.

The outer table will be the one without an index. The inner table will be the one with the index.

Assume the cost of each index probe is some constant value C per tuple.
Cost: $M + (m \times C)$

3 Sort-Merge Join

At a high-level, a sort-merge join sorts the two tables on their join key(s). The DBMS can use the external mergesort algorithm for this. It then steps through each of the tables with cursors and emits matches (like in mergesort).

This algorithm is useful if one or both tables are already sorted on join attribute(s) (like with a clustered index) or if the output needs to be sorted on the join key anyways.

The worst case scenario for this algorithm is if the join attribute for all the tuples in both tables contain the same value, which is very unlikely to happen in real databases. In this case, the cost of merging would be $M \cdot N$. Most of the time though, the keys are mostly unique so the merge cost is approximately $M + N$.

Assume that the DBMS has B buffers to use for the algorithm:

- Sort Cost for Table R: $2M \times 1 + \lceil log_{B-1} \lceil \frac{M}{B} \rceil \rceil$
- Sort Cost for Table S: $2N \times 1 + \lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil$
- Merge Cost: $(M + N)$

Total Cost: Sort + Merge

4 Hash Join

The high-level idea of the hash join algorithm is to use a hash table to split up the tuples into smaller chunks based on their join attribute(s). This reduces the number of comparisons that the DBMS needs to perform per tuple to compute the join. Hash join can only be used for equi-joins on the complete join key.

If tuple $r \in R$ and a tuple $s \in S$ satisfy the join condition, then they have the same value for the join attributes. If that value is hashed to some value i, the R tuple has to be in bucket r_i, and the S tuple has to be in bucket s_i. Thus, the R tuples in bucket r_i need only to be compared with the S tuples in bucket s_i.

Basic Hash Join

- **Phase #1 – Build:** First, scan the outer relation and populate a hash table using the hash function h_1 on the join attributes. The key in the hash table is the join attributes. The value depends on the implementation (can be full tuple values or a tuple id).
- **Phase #2 – Probe:** Scan the inner relation and use the hash function h_1 on each tuple's join attributes to jump to the corresponding location in the hash table and find a matching tuple. Since there may be collisions in the hash table, the DBMS will need to examine the original values of the join attribute(s) to determine whether tuples are truly matching.

If the DBMS knows the size of the outer table, the join can use a static hash table. If it does not know the size, then the join has to use a dynamic hash table or allow for overflow pages.

One optimization for the probe phase is the usage of a Bloom Filter. This is a probabalistic data structure that can fit in CPU caches and answer the question *is key x in the hash table?* with either *definitely no* or *probably yes*. This can reduce the amount of disk I/O by preventing disk reads that do not result in an emitted tuple.

Grace Hash Join / Hybrid Hash Join

When the tables do not fit on main memory, the DBMS has to swap tables in and out essentially at random, which leads to poor performance. The Grace Hash Join is an extension of the basic hash join that also hashes the inner table into partitions that are written out to disk.

- **Phase #1 – Build:** First, scan both the outer and inner tables and populate a hash table using the hash function h_1 on the join attributes. The hash table's buckets are written out to disk as needed. If a single bucket does not fit in memory, the DBMS can use *recursive partitioning* with different hash function h_2 (where $h_1 \neq h_2$) to further divide the bucket. This can continue recursively until the buckets fit into memory.
- **Phase #2 – Probe:** For each bucket level, retrieve the corresponding pages for both outer and inner tables. Then, perform a nested loop join on the tuples in those two pages. The pages will fit in memory, so this join operation will be fast.

Partitioning Phase Cost: $2 \times (M + N)$
Probe Phase Cost: $(M + N)$

Total Cost: $3 \times (M + N)$

5 Conclusion

Joins are an essential part of interacting with relational databases, and it is therefore critical to ensure that a DBMSs has efficient algorithms to execute joins.

JOIN ALGORITHM	I/O COST	TOTAL TIME
Simple Nested Loop Join	M + (m·N)	1.3 hours
Block Nested Loop Join	M + (M·N)	50 seconds
Index Nested Loop Join	M + (m·log N)	20 seconds
Sort Merge Join	M + N + (sort cost)	0.75 seconds

Figure 1: The table above assume the following: $M = 1000, m = 100000, N = 500, n = 40000$, and 0.1 ms/IO

Hash joins are almost always better than sort-based join algorithms, but there are cases in which sorting-based joins would be preferred. This includes queries on non-uniform data, when the data is already sorted on the join key, and when the result needs to be sorted. Good DBMSs will use either, or both.

Lecture #12: Query Processing I

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Andrew Crotty

1 Query Plan

The DBMS converts a SQL statement into a query plan. Operators in the query plan are arranged in a tree. Data flows from the leaves of this tree towards the root. The output of the root node in the tree is the result of the query. Typically operators are binary (1–2 children). The same query plan can be executed in multiple ways.

2 Processing Models

A DBMS *processing model* defines how the system executes a query plan. It specifies things like the direction in which the query plan is evaluated and what kind of data is passed between operators along the way. There are different models of processing models that have various trade-offs for different workloads.

These models can also be implemented to invoke the operators either from **top-to-bottom** or from **bottom-to-top**. Although the top-to-bottom approach is much more common, the bottom-to-top approach can allow for tighter control of caches/registers in *pipelines*.

The three execution models that we consider are:

- Iterator Model
- Materialization Model
- Vectorized / Batch Model

Iterator Model

The *iterator model*, also known as the Volcano or Pipeline model, is the most common processing model and is used by almost every (row-based) DBMS.

The iterator model works by implementing a `Next` function for every operator in the database. Each node in the query plan calls `Next` on its children until the leaf nodes are reached, which start emitting tuples up to their parent nodes for processing. Each tuple is then processed up the plan as far as possible before the next tuple is retrieved. This is useful in disk-based systems because it allows us to fully use each tuple in memory before the next tuple or page is accessed. A sample diagram of the iterator model is shown in Figure 1.

Every query plan operator implements a `Next` function as follows:

- On each call to `Next`, the operator returns either a single tuple or a null marker if there are no more tuples to emit.
- The operator implements a loop that calls `Next` on its children to retrieve their tuples and then process them. In this way, calling `Next` on a parent calls `Next` on its children. In response, the child node will return the next tuple that the parent must process.

The iterator model allows for *pipelining* where the DBMS can process a tuple through as many operators as possible before having to retrieve the next tuple. The series of tasks performed for a given tuple in the query plan is called a *pipeline*.

Some operators will block until children emit all of their tuples. Examples of such operators include joins, subqueries, and ordering (`ORDER BY`). Such operators are known as *pipeline breakers*.

Output control works easily with this approach (`LIMIT`) because an operator can stop invoking `Next` on its child (or children) operator(s) once it has all the tuples that it requires.

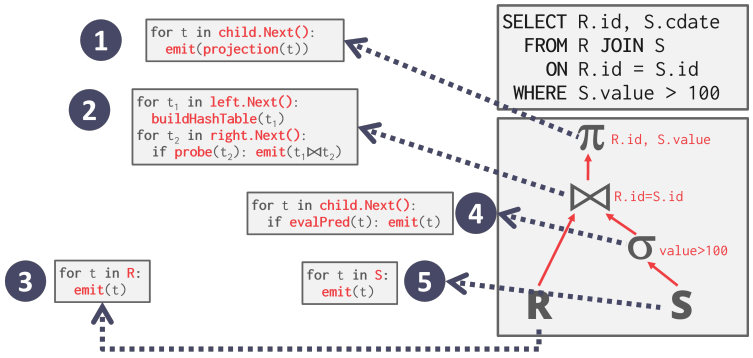


Figure 1: Iterator Model Example – Pseudo code of the different `Next` functions for each of the operators. The `Next` functions are essentially `for`-loops that iterate over the output of their child operator. For example, the root node calls `Next` on its child, the join operator, which is an access method that loops over the relation R and emits a tuple up that is then operated on. After all tuples have been processed, a null pointer (or another indicator) is sent that lets the parent nodes know to move on.

Materialization Model

The *materialization model* is a specialization of the iterator model where each operator processes its input all at once and then emits its output all at once. Instead of having a next function that returns a single tuple, each operator returns all of its tuples every time it is reached. To avoid scanning too many tuples, the DBMS can propagate down information about how many tuples are needed to subsequent operators (e.g. `LIMIT`). The operator "materializes" its output as a single result. The output can be either a whole tuple (NSM) or a subset of columns (DSM). A diagram of the materialization model is shown in Figure 2.

Every query plan operator implements an `Output` function:

- The operator processes all the tuples from its children at once.
- The return result of this function is all the tuples that operator will ever emit. When the operator finishes executing, the DBMS never needs to return to it to retrieve more data.

This approach is better for OLTP workloads because queries typically only access a small number of tuples at a time. Thus, there are fewer function calls to retrieve tuples. The materialization model is not suited for OLAP queries with large intermediate results because the DBMS may have to spill those results to disk between operators.

Vectorization Model

Like the iterator model, each operator in the *vectorization model* implements a `Next` function. However, each operator emits a *batch* (i.e. vector) of data instead of a single tuple. The operator's internal loop

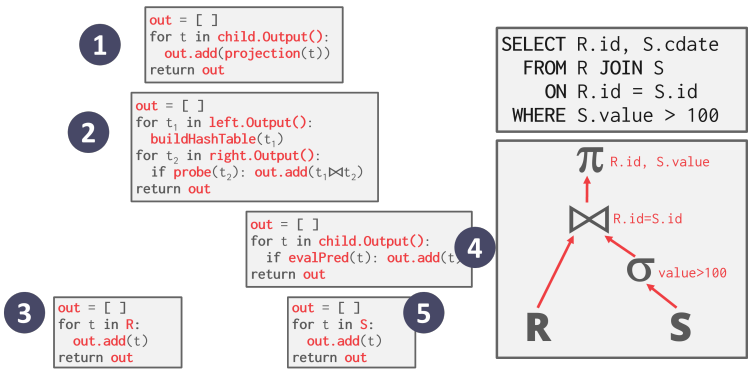


Figure 2: Materialization Model Example – Starting at the root, the `child.Output()` function is called, which invokes the operators below, which returns all tuples back up.

implementation is optimized for processing batches of data instead of a single item at a time. The size of the batch can vary based on hardware or query properties. See Figure 3 for an example of the vectorization model.

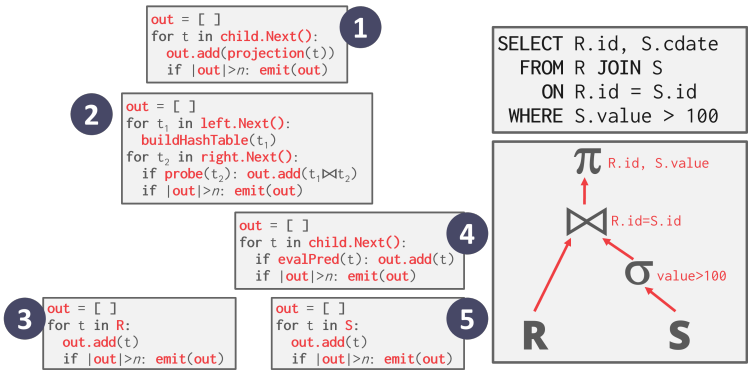


Figure 3: Vectorization Model Example – The vectorization model is very similar to the iterator model except at every operator, an output buffer is compared to the desired emission size. If the buffer is larger, then a tuple batch is sent up.

The vectorization model approach is ideal for OLAP queries that have to scan a large number of tuples because there are fewer invocations of the `Next` function.

The vectorization model allows operators to more easily use vectorized (SIMD) instructions to process batches of tuples.

Processing Direction

- **Approach #1: Top-to-Bottom**
  - Start with the root and "pull" data from children to parents
  - Tuples are always passed with function calls
- **Approach #2: Bottom-to-Top**
  - Start with leaf nodes and "push" data from children to parents
  - Allows for tighter control of caches / registers in operator pipelines

3 Access Methods

An *access method* is how the DBMS accesses the data stored in a table. In general, there are two approaches to access models; data is either read from a table or from an index with a sequential scan.

Sequential Scan

The sequential scan operator iterates over every page in the table and retrieves it from the buffer pool. As the scan iterates over all the tuples on each page, it evaluates the predicate to decide whether or not to emit the tuple to the next operator.

The DBMS maintains an internal cursor that tracks the last page/slot that it examined.

A sequential table scan is almost always the least efficient method by which a DBMS may execute a query. There are a number of optimizations available to help make sequential scans faster:

- **Prefetching:** Fetch the next few pages in advance so that the DBMS does not have to block on storage I/O when accessing each page.
- **Buffer Pool Bypass:** The scan operator stores pages that it fetches from disk in its local memory instead of the buffer pool in order to avoid sequential flooding.
- **Parallelization:** Execute the scan using multiple threads/processes in parallel.
- **Zone Map:** Pre-compute aggregations for each tuple attribute in a page. The DBMS can then decide whether it needs to access a page by checking its Zone Map first. The Zone Maps for each page are stored in separate pages and there are typically multiple entries in each Zone Map page. Thus, it is possible to reduce the total number of pages examined in a sequential scan. See figure Figure 4 for an example of a Zone Map.
- **Late Materialization:** DSM DBMSs can delay stitching together tuples until the upper parts of the query plan. This allows each operator to pass the minimal amount of information needed to the next operator (e.g. record ID, offset to record in column). This is only useful in column-store systems.
- **Heap Clustering:** Tuples are stored in the heap pages using an order specified by a clustering index.

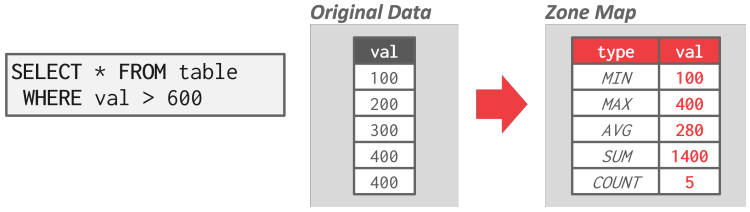


Figure 4: Zone Map Example – The zone map stores pre-computed aggregates for values in a page. In the example above, the select query realizes from the zone map that the max value in the original data is only 400. Then, instead of having to iterate through every tuple in the page, the query can avoid accessing the page at all since none of the values will be greater than 600.

Index Scan

In an *index scan*, the DBMS picks an index to find the tuples that a query needs.

There are many factors involved in the DBMSs' index selection process, including:

- What attributes the index contains
- What attributes the query references
- The attribute's value domains

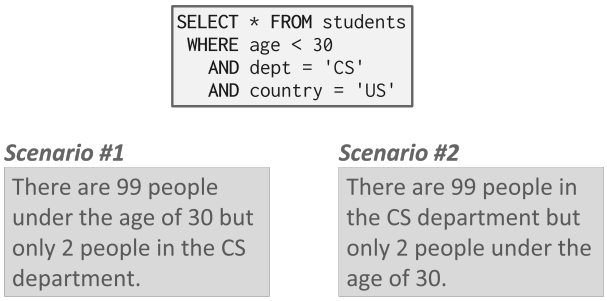


Figure 5: Index Scan Example – Consider a single table with 100 tuples and two indexes: `age` and `department`. In the first scenario, it is better to use the `department` index in the scan because it only has two tuples to match. Choosing the `age` index would not be much better than a simple sequential scan. In the second scenario, the `age` index would eliminate more unnecessary scans and is the optimal choice.

- Predicate composition
- Whether the index has unique or non-unique keys

A simple example of an index scan is shown in Figure 5.

More advanced DBMSs support multi-index scans. When using multiple indexes for a query, the DBMS computes sets of record IDs using each matching index, combines these sets based on the query's predicates, and retrieves the records and apply any predicates that may remain. The DBMS can use bitmaps, hash tables, or Bloom filters to compute record IDs through set intersection.

4 Modification Queries

Operators that modify the database (`INSERT`, `UPDATE`, `DELETE`) are responsible for checking constraints and updating indexes. For `UPDATE`/`DELETE`, child operators pass Record IDs for target tuples and must keep track of previously seen tuples.

There are two implementation choices on how to handle `INSERT` operators:

- **Choice #1:** Materialize tuples inside of the operator.
- **Choice #2:** Operator inserts any tuple passed in from child operators.

Halloween Problem

The Halloween Problem is an anomaly in which an update operation changes the physical location of a tuple, causing a scan operator to visit the tuple multiple times. This can occur on clustered tables or index scans.

This phenomenon was originally discovered by IBM researchers while building **System R** on Halloween day in 1976.

5 Expression Evaluation

The DBMS represents a `WHERE` clause as an *expression tree* (see Figure 6 for an example). The nodes in the tree represent different expression types.

Some examples of expression types that can be stored in tree nodes:

- Comparisons (=, <, >, !=)

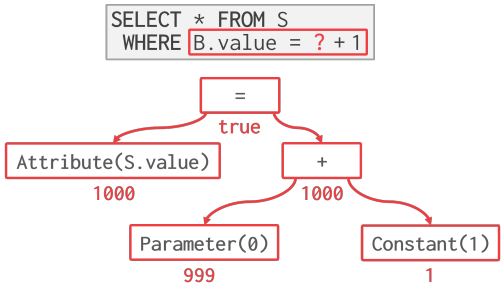


Figure 6: Expression Evaluation Example – A `WHERE` clause and a diagram of its corresponding expression.

- Conjunction (`AND`), Disjunction (`OR`)
- Arithmetic Operators (`+`, `-`, `*`, `/`, `%`)
- Constant and Parameter Values
- Tuple Attribute References

To evaluate an expression tree at runtime, the DBMS maintains a context handle that contains metadata for the execution, such as the current tuple, the parameters, and the table schema. The DBMS then walks the tree to evaluate its operators and produce a result.

Evaluating predicates in this manner is slow because the DBMS must traverse the entire tree and determine the correct action to take for each operator. A better approach is to just evaluate the expression directly.

Lecture #12: Query Execution II

15-445/645 Database Systems (Fall 2021)

https://15445.courses.cs.cmu.edu/fall2021/

Carnegie Mellon University
Andrew Crotty

1 Background

Previous discussions of query executions assumed that the queries executed with a single worker (i.e thread). However, in practice, queries are often executed in parallel with multiple workers.

Parallel execution provides a number of key benefits for DBMSs:

- Increased performance in throughput (more queries per second) and latency (less time per query).
- Increased responsiveness and availability from the perspective of external clients of the DBMS.
- Potentially lower *total cost of ownership* (TCO). This cost includes both the hardware procurement and software license, as well as the labor overhead of deploying the DBMS and the energy needed to run the machines.

There are two types of parallelism that DBMSs support: inter-query parallelism and intra-query parallelism.

2 Parallel vs Distributed Databases

In both parallel and distributed systems, the database is spread out across multiple "resources" to improve parallelism. These resources may be computational (e.g., CPU cores, CPU sockets, GPUs, additional machines) or storage (e.g., disks, memory).

It is important to distinguish between parallel and distributed systems.

- **Parallel DBMS** In a *parallel DBMS*, resources, or nodes, are physically close to each other. These nodes communicate with high-speed interconnect. It is assumed that communication between resources is not only fast, but also cheap and reliable.
- **Distributed DBMS** In a *distributed DBMS*, resources may be far away from each other; this might mean the database spans racks or data centers in different parts of the world. As a result, resources communicate using a slower interconnect over a public network. Communication costs between nodes are higher and failures cannot be ignored.

Even though a database may be physically divided over multiple resources, it still appears as a single logical database instance to the application. Thus, a SQL query executed against a single-node DBMS should generate the same result on a parallel or distributed DBMS.

3 Process Models

A DBMS *process model* defines how the system supports concurrent requests from a multi-user application/environment. The DBMS is comprised of more or more *workers* that are responsible for executing tasks on behalf of the client and returning the results. An application may send a large request or multiple requests at the same time that must be divided across different workers.

There are three different process models that a DBMS may adopt: process per worker, process pool, and thread per worker.

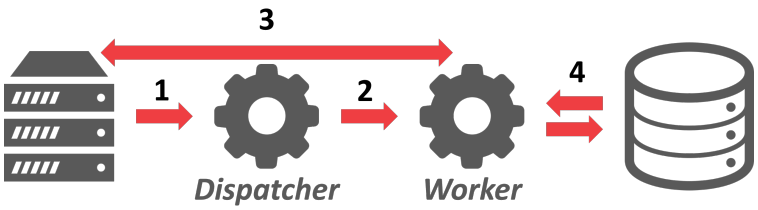


Figure 1: Process per Worker Model

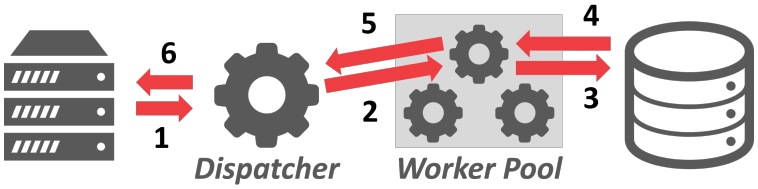


Figure 2: Process Pool Model

Process per Worker

The first and most basic approach is *process per worker*. Here, each worker is a separate OS process, and thus relies on OS scheduler. An application sends a request and opens a connection to the databases system. Some dispatcher receives the request and forks off a worker to handle this connection. The application now communicates directly with the worker who is responsible for executing the request that the query wants. This sequence of events is shown in Figure 1.

An advantage of the process per worker approach is that a process crash doesn't disrupt the whole system because each worker runs in the context of its own OS process.

This process model raises the issue of multiple workers on separate processes making numerous copies of the same page. A solution to maximize memory usage is to use shared-memory for global data structures so that they can be shared by workers running in different processes.

Examples of systems that utilize the process-per-worker process model include IBM DB2, Postgres, and Oracle.

Process Pool

An extension of the process per worker model is the *process pool* model. Instead of forking off processes for each connection request, workers are kept in a pool and selected by the dispatcher when a query arrives. Because the processes exist together in a pool, processes can share queries amongst themselves. A diagram of the process pool model is shown in Figure 2.

Like process per worker, the process pool also relies on the OS scheduler and shared memory.

A drawback to this approach is poor CPU cache locality as the same processes are not guaranteed to be used between queries.

Examples of systems that utilize the process pool process model include IBM DB2 and Postgres (post-2015).

Thread per Worker

The third and most common model is *thread per worker*. Instead of having different processes doing different tasks, each database system has only one process with multiple worker threads. In this environment,

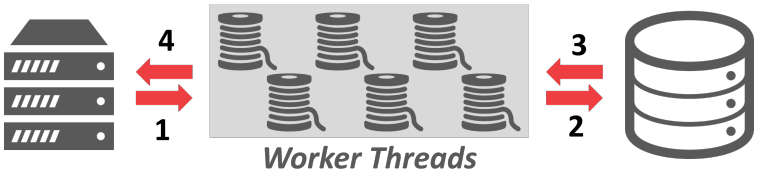


Figure 3: Thread per Worker Model

the DBMS has full control over the tasks and threads, it can manage it own scheduling. The multi-threaded model may or may not use a dispatcher thread. A diagram of the thread per worker model is shown in Figure 3.

Using multi-threaded architecture provides certain advantages. For one, there is less overhead per context switch. Additionally, a shared model does not have to be maintained. However, the thread per worker model does not necessarily imply that the DBMS supports intra-query parallelism.

Scheduling

In conclusion, for each query plan, the DBMS has to decide where, when, and how to execute. Relevant questions include:

- How many tasks should it use?
- How many CPU cores should it use?
- What CPU core should the tasks execute on?
- Where should a task store its output?

When making decisions regarding query plans, the DBMS **always** knows more than the OS and should be prioritized as such.

4 Inter-Query Parallelism

In *inter-query parallelism*, the DBMS executes different queries are concurrently. Because multiple workers are running requests simultaneously, overall performance is improved. This increases throughput and reduces latency.

If the queries are read-only, then little coordination is required between queries. However, if multiple queries are updating the database concurrently, more complicated conflicts arise. These issues are discussed further in lecture 15.

5 Intra-Query parallelism

In *intra-query parallelism*, the DBMS executes the operations of a single query in parallel. This decreases latency for long-running queries.

The organization of intra-query parallelism can be thought of in terms of a *producer/consumer* paradigm. Each operator is a producer of data as well as a consumer of data from some operator running below it.

Parallel algorithms exist for every relational operator. The DBMS can either have multiple threads access centralized data structures or use partitioning to divide work up.

Within intra-query parallelism, there are three types of parallelism: intra-operator, inter-operator, and bushy. These approaches are not mutually exclusive. It is the DBMS' responsibility to combine these techniques in a way that optimizes performance on a given workload.

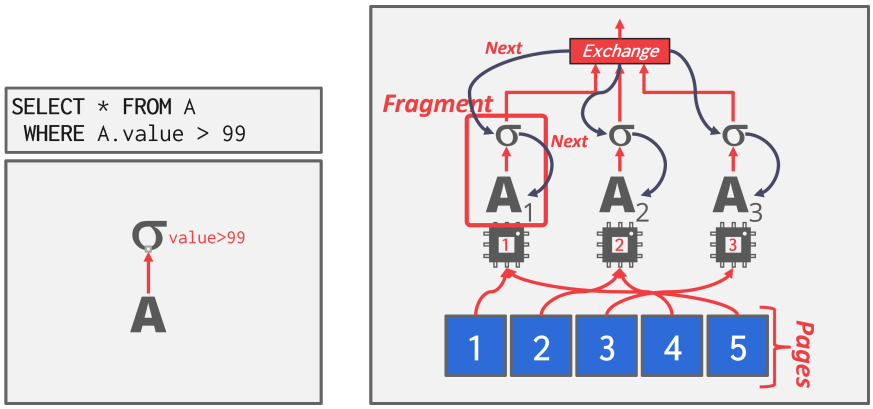


Figure 4: Intra-Operator Parallelism – The query plan for this `SELECT` is a sequential scan on A that is fed into a filter operator. To run this in parallel, the query plan is partitioned into disjoint fragments. A given plan fragment is operated on a by a distinct worker. The exchange operator calls `Next` concurrently on all fragments which then retrieve data from their respective pages.

Intra-Operator Parallelism (Horizontal)

In *intra-operator parallelism*, the query plan's operators are decomposed into independent *fragments* that perform the same function on different (disjoint) subsets of data.

The DBMS inserts an *exchange* operator into the query plan to coalesce results from child operators. The exchange operator prevents the DBMS from executing operators above it in the plan until it receives all of the data from the children. An example of this is shown in Figure 4.

In general, there are three types of exchange operators:

- **Gather:** Combine the results from multiple workers into a single output stream. This is the most common type used in parallel DBMSs.
- **Repartition:** Reorganize multiple input streams across multiple output streams. This allows the DBMS take inputs that are partitioned one way and then redistribute them in another way.
- **Distribute:** Split a single input stream into multiple output streams.

Inter-Operator Parallelism (Vertical)

In *inter-operator parallelism*, the DBMS overlaps operators in order to pipeline data from one stage to the next without materialization. This is sometimes called *pipelined parallelism*. See example in Figure 5.

This approach is widely used in *stream processing systems*, which are systems that continually execute a query over a stream of input tuples.

Bushy Parallelism

Bushy parallelism is a hybrid of intra-operator and inter-operator parallelism where workers execute multiple operators from different segments of the query plan at the same time.

The DBMS still uses exchange operators to combine intermediate results from these segments. An example is shown in Figure 6.

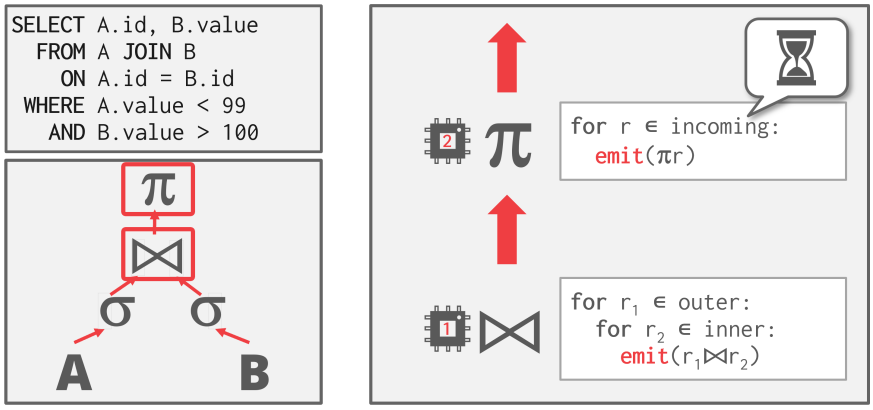


Figure 5: Inter-operator Parallelism – In the JOIN statement to the left, a single worker performs the join and then emits the result to another worker that performs the projection and then emits the result again.

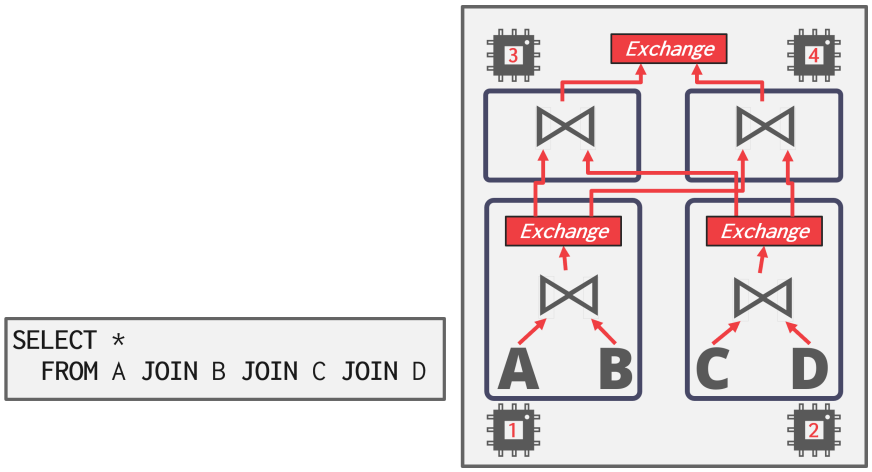


Figure 6: Bushy Parallelism – To perform a 4-way JOIN on three tables, the query plan is divided into four fragments as shown. Different portions of the query plan run at the same time, in a manner similar to inter-operator parallelism.

6 I/O Parallelism

Using additional processes/threads to execute queries in parallel will not improve performance if the disk is always the main bottleneck. Therefore, it is important to be able to split a database across multiple storage devices.

To get around this, DBMSs use I/O parallelism to *split installation across multiple devices*. Two approaches to I/O parallelism are multi-disk parallelism and database partitioning.

Multi-Disk Parallelism

In *multi-disk parallelism*, the OS/hardware is configured to store the DBMS's files across multiple storage devices. This can be done through storage appliances or RAID configuration. All of the storage setup is transparent to the DBMS so workers cannot operate on different devices because the DBMS is unaware of the underlying parallelism.

Database Partitioning

In *database partitioning*, the database is split up into disjoint subsets that can be assigned to discrete disks. Some DBMSs allow for specification of the disk location of each individual database. This is easy to do at the file-system level if the DBMS stores each database in a separate directory. The log file of changes made is usually shared.

The idea of *logical partitioning* is to split single logical table into disjoint physical segments that are stored/managed separately. Such partitioning is ideally transparent to the application. That is, the application should be able to access logical tables without caring how things are stored.

The two approaches to partitioning are vertical and horizontal partitioning.

In *vertical partitioning*, a table's attributes are stored in a separate location (like a column store). The tuple information must be stored in order to reconstruct the original record.

In *horizontal partitioning*, the tuples of a table are divided into disjoint segments based on some partitioning keys. There are different ways to decide how to partition (e.g., hash, range, or predicate partitioning). The efficacy of each approach depends on the queries.

Lecture #13: Query Planning & Optimization I

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Overview

Because SQL is declarative, the query only tells the DBMS what to compute, but not how to compute it. Thus, the DBMS needs to translate a SQL statement into an executable query plan. But there are different ways to execute each operator in a query plan (e.g., join algorithms) and there will be differences in performance among these plans. The job of the DBMS's optimizer is to pick an optimal plan for any given query.

The first implementation of a query optimizer was IBM System R and was designed in the 1970s. Prior to this, people did not believe that a DBMS could ever construct a query plan better than a human. Many concepts and design decisions from the System R optimizer are still in use today.

There are two high-level strategies for query optimization.

The first approach is to use static rules, or *heuristics*. Heuristics match portions of the query with known patterns to assemble a plan. These rules transform the query to remove inefficiencies. Although these rules may require consultation of the catalog to understand the structure of the data, they never need to examine the data itself.

An alternative approach is to use *cost-based search* to read the data and estimate the cost of executing equivalent plans. The cost model chooses the plan with the lowest cost.

Query optimization is the most difficult part of building a DBMS. Some systems have attempted to apply machine learning to improve the accuracy and efficiency of optimizers, but no major DBMS currently deploys an optimizer based on this technique.

Logical vs. Physical Plans

The optimizer generates a mapping of a *logical algebra expression* to the optimal equivalent physical algebra expression. The logical plan is roughly equivalent to the relational algebra expressions in the query.

Physical operators define a specific execution strategy using an access path for the different operators in the query plan. Physical plans may depend on the physical format of the data that is processed (i.e. sorting, compression).

There does not always exist a one-to-one mapping from logical to physical plans.

2 Relational Algebra Equivalence

Much of query optimization relies on the underlying concept that the high level properties of relational algebra are preserved across equivalent expressions. Two relational algebra expressions are *equivalent* if they generate the same set of tuples.

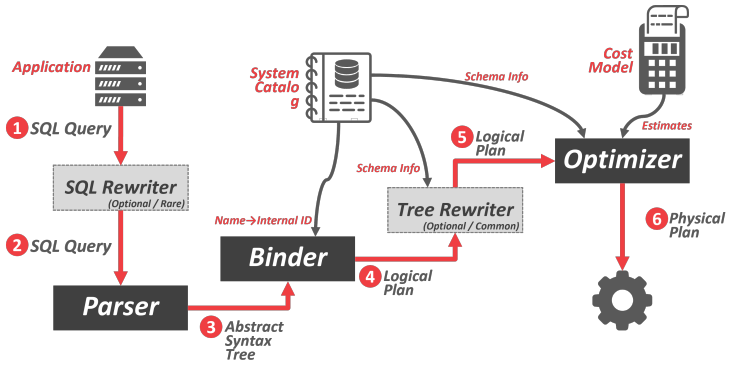


Figure 1: Architecture Overview – The application connected to the database system and sends a SQL query, which may be rewritten to a different format. The SQL string is parsed into tokens that make up the syntax tree. The binder converts named objects in the syntax tree to internal identifiers by consulting the system catalog. The binder emits a logical plan which may be fed to a tree rewriter for additional schema info. The logical plan is given to the optimizer which selects the most efficient procedure to execute the plan.

This technique of transforming the underlying relational algebra representation of a logical plan is known as *query rewriting*.

One example of relational algebra equivalence is *predicate pushdown*, in which a predicate is applied in a different position of the sequence to avoid unnecessary work. Figure 2 shows an example of predicate pushdown.

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```

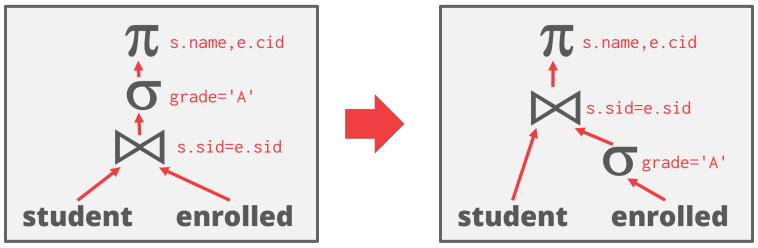


Figure 2: Predicate Pushdown: – Instead of performing the filter after the join, the filter can be applied earlier in order to pass fewer elements into the filter.

3 Logical Query Optimization

Some selection optimizations include:

- Perform filters as early as possible (predicate pushdown).
- Reorder predicates so that the DBMS applies the most selective one first.
- Breakup a complex predicate and pushing it down (split conjunctive predicates).

An example of predicate pushdown is shown in Figure 2.

Some projection optimizations include:

- Perform projections as early as possible to create smaller tuples and reduce intermediate results *(projection pushdown)*.
- Project out all attributes except the ones requested or requires.

An example of projection pushdown in shown in Figure 3.

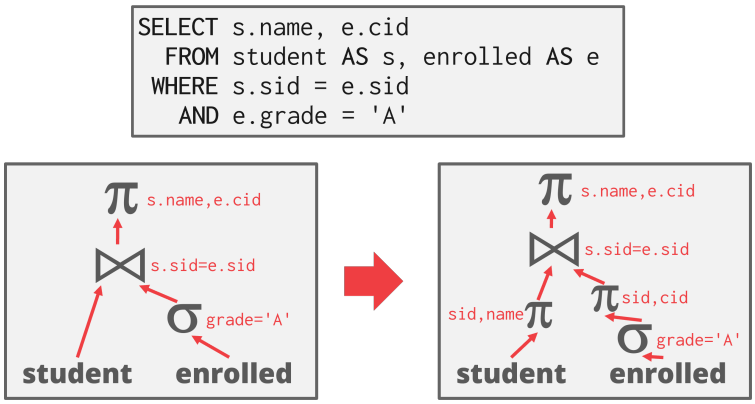


Figure 3: Projection Pushdown – Since the query only asks for the student name and ID, the DBMS can remove all columns except for those two before applying the join.

Another optimization that a DBMS can use is to remove impossible or *unnecessary predicates*. In this optimization, the DBMS elides evaluation of predicates whose result does not change per tuple in a table. Bypassing these predicates reduces computation cost. Figure 4 shows two examples of unnecessary predicates.

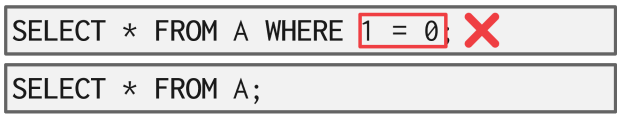


Figure 4: Unnecessary Predicates – The predicate in the first query will always be false and can be disregarded. The former query can be rewritten as the latter query to produce the same result but save on computation.

A similar optimization is *merging predicates*. An example of this optimization is shown in Figure 5.

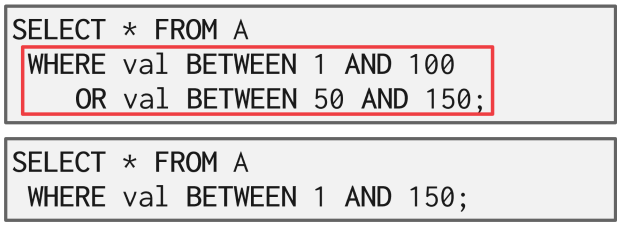


Figure 5: Merging Predicates – The `WHERE` predicate in query 1 has redundancy as what it is searching for is any value between 1 and 150. Query 2 shows the more succinct way to express request in query 1.

The ordering of JOIN operations is a key determinant of query performance. Exhaustive enumeration of all possible join orders is inefficient, so join-ordering optimization requires a cost model. However, we can still eliminate *unnecessary joins* with a heuristic approach to optimization. An example of join elimination is shown in Figure 6.

```
SELECT A1.*
  FROM A AS A1 JOIN A AS A2
    ON A1.id = A2.id;
```

```
SELECT * FROM A;
```

Figure 6: Join Elimination – The join in query 1 is wasteful because every tuple in A must exist in A. Query 1 can instead be written as query 2.

The DBMS can also optimize nested sub-queries without referencing a cost model. There are two different approaches to this type of optimization:

- Re-write the query by de-correlating and / or flattening it. An example of this is shown in Figure 7.
- Decompose the nested query and store the result to a temporary table. An example of this is shown in Figure 8.

```
SELECT name FROM sailors AS S
 WHERE EXISTS (
    SELECT * FROM reserves AS R
     WHERE S.sid = R.sid
       AND R.day = '2018-10-15'
 )
```

```
SELECT name
  FROM sailors AS S, reserves AS R
 WHERE S.sid = R.sid
   AND R.day = '2018-10-15'
```

Figure 7: Subquery Optimization - Rewriting The former query can be rewritten as the latter query by rewriting the subquery as a JOIN. Removing a level of nesting in this way effectively *flattens* the query.

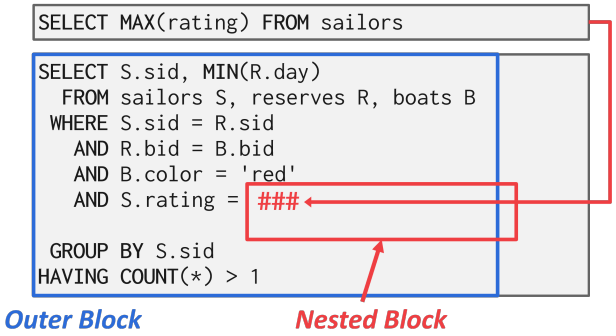


Figure 8: Subquery Optimization - Decomposition – For complex queries with subqueries, the DBMS optimizer may break up the original query into blocks and focus on optimizing each individual block at a a time. In this example, the optimizer decomposes a query with a nested aggregation by pulling the nested query out into its own query, and subsequently using this result to realize the logic of the original query.

Lecture #14: Query Planning & Optimization II

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Cost Estimations

DBMS's use cost models to estimate the cost of executing a plan. These models evaluate equivalent plans for a query to help the DBMS select the most optimal one.

The cost of a query depends on several underlying metrics, including:

- **CPU:** small cost, but tough to estimate.
- **Disk I/O:** the number of block transfers.
- **Memory:** the amount of DRAM used.
- **Network:** the number of messages sent.

Exhaustive enumeration of all valid plans for a query is much too slow for an optimizer to perform. For joins alone, which are commutative and associative, there are 4^n different orderings of every n-way join. Optimizers must limit their search space in order to work efficiently.

To approximate costs of queries, DBMS's maintain internal *statistics* about tables, attributes, and indexes in their internal catalogs. Different systems maintain these statistics in different ways. Most systems attempt to avoid on-the-fly computation by maintaining an internal table of statistics. These internal tables may then be updated in the background.

For each relation R, the DBMS maintains the following information:

- N_R: Number of tuples in R
- $V(A, R)$: Number of distinct values of attribute A

With the information listed above, the optimizer can derive the *selection cardinality* $SC(A, R)$ statistic. The selection cardinality is the average number of records with a value for an attribute A given $\frac{N_R}{V(A,R)}$. Note that this assumes data uniformity. This assumption is often incorrect, but it simplifies the optimization process.

Selection Statistics

The selection cardinality can be used to determine the number of tuples that will be selected for a given input.

Equality predicates on unique keys are simple to estimate (see Figure 1). A more complex predicate is shown in Figure 2.

The *selectivity* (sel) of a predicate P is the fraction of tuples that qualify. The formula used to compute selective depends on the type of predicate. Selectivity for complex predicates is hard to estimate accurately which can pose a problem for certain systems. An example of a selectivity computation is shown in Figure 3.

```
SELECT * FROM people
 WHERE id = 123
```

```
CREATE TABLE people (
  id INT PRIMARY KEY,
  val INT NOT NULL,
  age INT NOT NULL,
  status VARCHAR(16)
);
```

Figure 1: Simple Predicate Example – In this example, determining what index to use is easy because the query contains an equality predicate on a unique key.

```
SELECT * FROM people
 WHERE val > 1000
```

```
SELECT * FROM people
 WHERE age = 30
   AND status = 'Lit'
   AND age+id IN (1,2,3)
```

Figure 2: Complex Predicate Example – More complex predicates, such as range or conjunctions, are harder to estimate because the selection cardinalities of the predicates must be combined in non-trivial ways.

Negation Query:
→ sel(not P) = 1 - sel(P)
→ Example: sel(age != 2) = 1 - (1/5) = 4/5

```
SELECT * FROM people
 WHERE age != 2
```

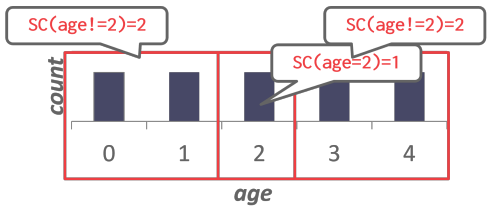


Figure 3: Selectivity of Negation Query Example – The selectivity of the negation query is computed by subtracting the selectivity of the positive query from 1. In the example, the answer comes out to be $\frac{4}{5}$ which is accurate.

Observe that the selectivity of a predicate is equivalent to the probability of that predicate. This allows probability rules to be applied in many selectivity computations. This is particularly useful when dealing with complex predicates. For example, if we assume that multiple predicates involved in a conjunction are *independent*, we can compute the total selectivity of the conjunction as the product of the selectivities of the individual predicates.

Selectivity Computation Assumptions

In computing the selection cardinality of predicates, the following three assumptions are used.

- **Uniform Data**: The distribution of values (except for the heavy hitters) is the same.
- **Independent Predicates**: The predicates on attributes are independent.
- **Inclusion Principle**: The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.

These assumptions are often not satisfied by real data. For example, *correlated attributes* break the assumption of independence of predicates.

2 Histograms

Real data is often skewed and is tricky to make assumptions about. However, storing every single value of a data set is expensive. One way to reduce the amount of memory used by storing data in a *histogram* to group together values. An example of a graph with buckets is shown in Figure 4.

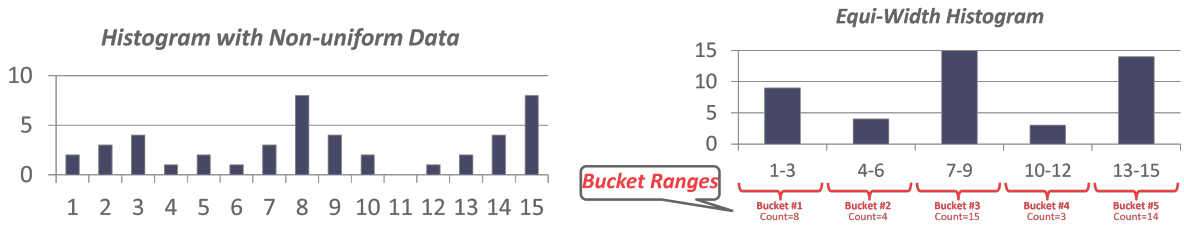


Figure 4: Equi-Width Histogram: The first figure shows the original frequency count of the entire data set. The second figure is an equi-width histogram that combines together the counts for adjacent keys to reduce the storage overhead.

Another approach is to use a *equi-depth* histogram that varies the width of buckets so that the total number of occurrences for each bucket is roughly the same. An example is shown in Figure 5.

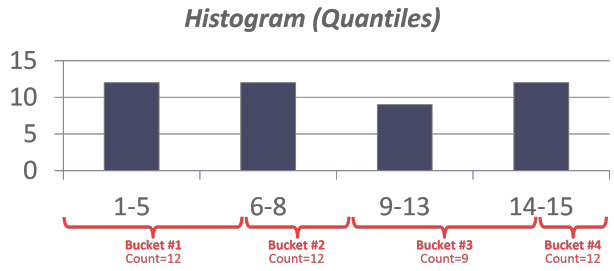


Figure 5: Equi-Depth Histogram – To ensure that each bucket has roughly the same number of counts, the histogram varies the range of each bucket.

In place of histograms, some systems may use *sketches* to generate approximate statistics about a data set.

3 Sampling

DBMS's can use *sampling* to apply predicates to a smaller copy of the table with a similar distribution (see Figure 6). The DBMS updates the sample whenever the amount of changes to the underlying table exceeds some threshold (e.g., 10% of the tuples).

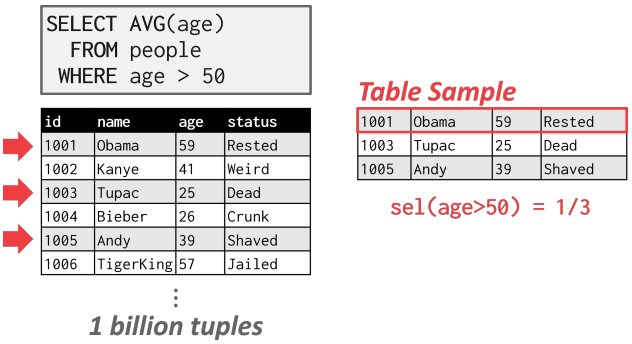


Figure 6: Sampling – Instead of using one billion values in the table to estimate selectivity, the DBMS can derive the selectivities for predicates from a subset of the original table.

4 Plan Enumeration

After performing rule-based rewriting, the DBMS will enumerate different plans for the query and estimate their costs. It then chooses the best plan for the query after exhausting all plans or some timeout.

Single-Relation Query Plans

For single-relation query plans, the biggest obstacle is choosing the best access method (i.e., sequential scan, binary search, index scan, etc.) Most new database systems just use heuristics, instead of a sophisticated cost model, to pick an access method.

For OLTP queries, this is especially easy because they are *sargable* (Search Argument Able), which means that there exists a best index that can be selected for the query. This can also be implemented with simple heuristics.

Multi-Relation Query Plans

As the number of joins increases, the number of alternative plans grows rapidly. To deal with this, we need to restrict the search space. **IBM System R** made the fundamental decision to only consider left-deep join trees (see Figure 7). This is because left-deep join trees are better suited for the pipeline model since the the DBMS does not need to materialize the outputs of the join operators. If the DBMS's optimizer only considers left-deep trees, then it will reduce the amount of memory that the search processes uses and potentially reduce the search time. Most modern DBMSs do not make this restriction during optimization.

Figure 7: System R Optimizer – The first cost-based query optimizer in **IBM System R** only considered left-deep join trees.

To make query plans, the DBMS must first enumerate the orderings, then the plans for each operator, followed by the access paths for each table. See Figure 8 for an example. *Dynamic programming* can be used to reduce the number of cost estimations.

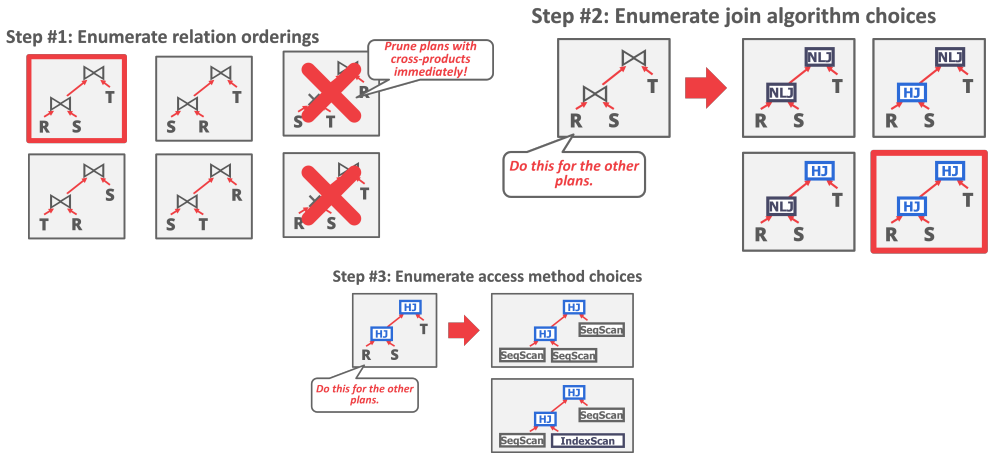


Figure 8: Candidate Plans Example – The first step is to enumerate all relation orderings. Any orderings with cross-products or non-left deep joins can be pruned. In the second step, all join algorithm choices (e.g. nested loop join, hash join, sort-merge join) are enumerated. In step three, the access methods are enumerated to find the cheapest path.

Lecture #15: Concurrency Control Theory

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Transactions

A *transaction* is the execution of a sequence of one or more operations (e.g., SQL queries) on a shared database to perform some higher level function. They are the basic unit of change in a DBMS. Partial transactions are not allowed (i.e. transactions must be atomic).

Example: Move $100 from Lin's bank account to his promotor's account

1. Check whether Lin has $100.
2. Deduct $100 from his account.
3. Add $100 to his promotor's account.

Either all of the steps need to be completed or none of them should be completed.

The Strawman System

A simple system for handling transactions is to execute one transaction at a time usinga single worker (e.g. one thread). Thus, only one transaction can be running at a time. To execute the transaction, the DBMS copies the entire database file and makes the transaction changes to this new file. If the transaction succeeds, then the new file becomes the current database file. If the transaction fails, the DBMS discards the new file and none of the transaction's changes have been saved. This method is slow as it does not allow for concurrent transactions and requires copying the whole database file for every transaction.

A (potentially) better approach is to allow concurrent execution of independent transactions while also maintaining correctness and fairness (as in all transactions are treated with equal priority and don't get "starved" by never being executed). But executing concurrent transactions in a DBMS is challenging. It is difficult to ensure correctness (for example, if Lin only has $100 and tries to pay off two promoters at once, who should get paid?) while also executing transactions quickly (our strawman example guarantees sequential correctness, but at the cost of parallelism).

Arbitrary interleaving of operations can lead to:

- **Temporary Inconsistency:** Unavoidable, but not an issue.
- **Permanent Inconsistency:** Unacceptable, cause problems with correctness and integrity of data.

The scope of a transaction is only inside the database. It cannot make changes to the outside world because it cannot roll those back. For example, if a transaction causes an email to be sent, this cannot be rolled back by the DBMS if the transaction is aborted.

2 Definitions

Formally, a *database* can be represented as a set of named data objects ($A, B, C, \ldots$). These objects can be attributes, tuples, pages, tables, or even databases. The algorithms that we will discuss work on any type of object but all objects must be of the same type.

A *transaction* is a sequence of read and write operations (i.e., $R(A), W(B)$) on those objects. To simplify discurssion, this definition assumes the database is a fixed size, so the operations can only be reads and updates, not inserts or deletions.

The boundaries of transactions are defined by the client. In SQL, a transaction starts with the `BEGIN` command. The outcome of a transaction is either `COMMIT` or `ABORT`. For `COMMIT`, either all of the transaction's modifications are saved to the database, or the DBMS overrides this and aborts instead.

For `ABORT`, all of the transaction's changes are undone so that it is like the transaction never happened. Aborts can be either self-inflicted or caused by the DBMS.

The criteria used to ensure the correctness of a database is given by the acronym **ACID**.

- **A**tomicity: Atomicity ensures that either all actions in the transaction happen, or none happen.
- **C**onsistency: If each transaction is consistent and the database is consistent at the beginning of the transaction, then the database is guaranteed to be consistent when the transaction completes.
- **I**solation: Isolation means that when a transaction executes, it should have the illusion that it is isolated from other transactions.
- **D**urability: If a transaction commits, then its effects on the database should persist.

3 ACID: Atomicity

The DBMS guarantees that transactions are **atomic**. The transaction either executes all its actions or none of them. There are two approaches to this:

Approach #1: Logging
DBMS logs all actions so that it can undo the actions of aborted transactions. It maintains undo records both in memory and on disk. Logging is used by almost all modern systems for audit and efficiency reasons.

Approach #2: Shadow Paging
The DBMS makes copies of pages modified by the transactions and transactions make changes to those copies. Only when the transaction commits is the page made visible. This approach is typically slower at runtime than a logging-based DBMS. However, one benefit is, if you are only single threaded, there is no need for logging, so there are less writes to disk when transactions modify the database. This also makes recovery simple, as all you need to do is delete all pages from uncommitted transactions. In general, though, better runtime performance is preferred over better recovery performance, so this is rarely used in practice.

4 ACID: Consistency

At a high level, consisitency means the "world" represented by the database is **logically** correct. All questions (i.e., queries) that the application asks about the data will return logically correct results. There are two notions of consistency:

Database Consistency: The database accurately represents the real world entity it is modeling and follows integrity constraints. (E.g. The age of a person cannot not be negative). Additionally, transactions in the future should see the effects of transactions committed in the past inside of the database.

Transaction Consistency: If the database is consistent before the transaction starts, it will also be consistent after. Ensuring transaction consistency is the application's responsibility.

5 ACID: Isolation

The DBMS provides transactions the illusion that they are running alone in the system. They do not see the effects of concurrent transactions. This is equivalent to a system where transactions are executed in serial order (i.e., one at a time). But to achieve better performance, the DBMS has to interleave the operations of concurrent transactions while maintaining the illusion of isolation.

Concurrency Control

A *concurrency control protocol* is how the DBMS decides the proper interleaving of operations from multiple transactions at runtime.

There are two categories of concurrency control protocols:

1. **Pessimistic:** The DBMS assumes that transactions will conflict, so it doesn't let problems arise in the first place.
2. **Optimistic:** The DBMS assumes that conflicts between transactions are rare, so it chooses to deal with conflicts when they happen after the transactions commit.

The order in which the DBMS executes operations is called an *execution schedule*. The goal of a concurrency control protocol is to generate an execution schedule that is is equivalent to some serial execution:

- **Serial Schedule:** Schedule that does not interleave the actions of different transactions.
- **Equivalent Schedules:** For any database state, if the effect of execution the first schedule is identical to the effect of executing the second schedule, the two schedules are equivalent.
- **Serializable Schedule**: A serializable schedule is a schedule that is equivalent to any serial execution of the transactions. Different serial executions can produce different results, but all are considered "correct".

A *conflict* between two operations occurs if the operations are for different transactions, they are performed on the same object, and at least one of the operations is a write. There are three variations of conflicts:

- **Read-Write Conflicts ("Unrepeatable Reads"):** A transaction is not able to get the same value when reading the same object multiple times.
- **Write-Read Conflicts ("Dirty Reads"):** A transaction sees the write effects of a different transaction before that transaction committed its changes.
- **Write-Write conflict ("Lost Updates"):** One transaction overwrites the uncommitted data of another concurrent transaction.

There are two types for serializability: (1) *conflict* and (2) *view*. Neither definition allows all schedules that one would consider serializable. In practice, DBMSs support conflict serializability because it can be enforced efficiently.

Conflict Serializability

Two schedules are *conflict equivalent* if they involve the same operations of the same transactions and every pair of conflicting operations is ordered in the same way in both schedules. A schedule S is *conflict serializable* if it is conflict equivalent to some serial schedule.

One can verify that a schedule is conflict serializable by swapping non-conflicting operations until a serial schedule is formed. For schedules with many transactions, this becomes too expensive. A better way to verify schedules is to use a *dependency graph* (precedence graph).

In a dependency graph, each transaction is a node in the graph. There exists a directed edge from node T_i to T_j iff an operation O_i from T_i conflicts with an operation O_j from T_j and O_i occurs before O_j in the

schedule. Then, a schedule is conflict serializable iff the dependency graph is acyclic.

View Serializability

View serializability is a weaker notion of serializibility that allows for all schedules that are conflict serializable and "blind writes" (i.e. performing writes without reading the value first). Thus, it allows for more schedules than conflict serializability, but is difficult to enforce efficiently. This is because the DBMS does not know how the application will "interpret" values.

6 ACID: Durability

All of the changes of committed transactions must be **durable** (i.e., persistent) after a crash or restart. The DBMS can either use logging or shadow paging to ensure that all changes are durable.

Lecture #16: Two-Phase Locking

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Transaction Locks

A DBMS uses *locks* to dynamically generate an execution schedule for transactions that is serializable without knowing each transaction's read/write set ahead of time. These locks protect database objects during concurrent access when there are multiple readers and writes. The DBMS contains a centralized *lock manager* that decides whether a transaction can acquire a lock or not. It also provides a global view of whats going on inside the system.

There are two basic types of locks:

- **Shared Lock (`S-LOCK`):** A shared lock that allows multiple transactions to read the same object at the same time. If one transaction holds a shared lock, then another transaction can also acquire that same shared lock.
- **Exclusive Lock (`X-LOCK`):** An exclusive lock allows a transaction to modify an object. This lock prevents other transactions from taking any other lock (`S-LOCK` or `X-LOCK`) on the object. Only one transaction can hold an exclusive lock at a time.

Transactions must request locks (or upgrades) from the lock manager. The lock manager grants or blocks requests based on what locks are currently held by other transactions. Transactions must release locks when they no longer need them to free up the object. The lock manager updates its internal lock-table with information about which transactions hold which locks and which transactions are waiting to acquire locks.

The DBMS's lock-table does not need to be durable since any transaction that is active (i.e., still running) when the DBMS crashes is automatically aborted.

2 Two-Phase Locking

Two-Phase locking (2PL) is a pessimistic concurrency control protocol that uses locks to determine whether a transaction is allowed to access an object in the database on the fly. The protocol does not need to know all of the queries that a transaction will execute ahead of time.

Phase #1– Growing: In the growing phase, each transaction requests the locks that it needs from the DBMS's lock manager. The lock manager grants/denies these lock requests.

Phase #2– Shrinking: Transactions enter the shrinking phase immediately after it releases its first lock. In the shrinking phase, transactions are only allowed to release locks. They are not allowed to acquire new ones.

On its own, 2PL is sufficient to guarantee conflict serializability. It generates schedules whose precedence graph is acyclic. But it is susceptible to *cascading aborts*, which is when a transaction aborts and now another transaction must be rolled back, which results in wasted work.

There are also potential schedules that are serializable but would not be allowed by 2PL (locking can limit concurrency).

Strong Strict Two-Phase Locking

A schedule is *strict* if any value written by a transaction is never read or overwritten by another transaction until the first transaction commits. *Strong Strict 2PL* (also known as *Rigorous 2PL*) is a variant of 2PL where the transactions only release locks when they commit.

The advantage of this approach is that the DBMS does not incur cascading aborts. The DBMS can also reverse the changes of an aborted transaction by restoring the original values of modified tuples. However, Strict 2PL generates more cautious/pessimistic schedules that limit concurrency.

3 Deadlock Handling

A *deadlock* is a cycle of transactions waiting for locks to be released by each other. There are two approaches to handling deadlocks in 2PL: detection and prevention.

Approach #1: Deadlock Detection

To detect deadlocks, the DBMS creates a *waits-for* graph where transactions are nodes, and there exists a directed edge from T_i to T_j if transaction T_i is waiting for transaction T_j to release a lock. The system will periodically check for cycles in the waits-for graph (usually with a background thread) and then make a decision on how to break it. Latches are not needed when constructing the graph since if the DBMS misses a deadlock in one pass, it will find it in the subsequent passes.

When the DBMS detects a deadlock, it will select a "victim" transaction to abort to break the cycle. The victim transaction will either restart or abort depending on how the application invoked it.

The DBMS can consider multiple transaction properties when selecting a victim to break the deadlock:

1. By age (newest or oldest timestamp).
2. By progress (least/most queries executed).
3. By the # of items already locked.
4. By the # of transactions needed to rollback with it.
5. # of times a transaction has been restarted in the past (to avoid starvation).

There is no one choice that is better than others. Many systems use a combination of these factors.

After selecting a victim transaction to abort, the DBMS can also decide on how far to rollback the transaction's changes. It can either rollback the entire transaction or just enough queries to break the deadlock.

Approach #2: Deadlock Prevention

Instead of letting transactions try to acquire any lock they need and then deal with deadlocks afterwards, deadlock prevention 2PL stops transactions from causing deadlocks before they occur. When a transaction tries to acquire a lock held by another transaction (which could cause a deadlock), the DBMS kills one of them. To implement this, transactions are assigned priorities based on timestamps (older transactions have higher priority). These schemes guarantee no deadlocks because only one type of direction is allowed when waiting for a lock. When a transaction restarts, the DBMS reuses the same timestamp.

There are two ways to kill transactions under deadlock prevention:

- **Wait-Die ("Old Waits for Young"):** If the requesting transaction has a higher priority than the holding transaction, it waits. Otherwise, it aborts.
- **Wound-Wait ("Young Waits for Old"):** If the requesting transaction has a higher priority than the holding transaction, the holding transaction aborts and releases the lock. Otherwise, the requesting transaction waits.

4 Lock Granularities

If a transaction wants to update one billion tuples, it has to ask the DBMS's lock manager for a billion locks. This will be slow because the transaction has to take latches in the lock manager's internal lock table data structure as it acquires/releases locks.

To avoid this overhead, the DBMS can use to use a lock hierarchy that allows a transaction to take more coarse-grained locks in the system. For example, it could acquire a single lock on the table with one billion tuples instead of one billion separate locks. When a transaction acquires a lock for an object in this hierarchy, it implicitly acquires the locks for all its children objects.

Intention locks allow a higher level node to be locked in shared mode or exclusive mode without having to check all descendant nodes. If a node is in an intention mode, then explicit locking is being done at a lower level in the tree.

- **Intention-Shared (`IS`):** Indicates explicit locking at a lower level with shared locks.
- **Intention-Exclusive (`IX`):** Indicates explicit locking at a lower level with exclusive or shared locks.
- **Shared+Intention-Exclusive (`SIX`):** The sub-tree rooted at that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

Lecture #17: Timestamp Ordering Concurrency Control

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Timestamp Ordering Concurrency Control

Timestamp ordering (T/O) is an optimistic class of concurrency control protocols where the DBMS assumes that transaction conflicts are rare. Instead of requiring transactions to acquire locks before they are allowed to read/write to a database object, the DBMS instead uses timestamps to determine the serializability order of transactions.

Each transaction T_i is assigned a unique fixed timestamp TS(T_i) that is monotonically increasing. Different schemes assign timestamps at different times during the transaction. Some advanced schemes even assign multiple timestamps per transaction.

If TS(T_i) < TS(T_j), then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where T_i appears before T_j.

There are multiple timestamp allocation implementation strategies. The DBMS can use the system clock as a timestamp, but issues arise with edge cases like daylight savings. Another option is to use a logical counter. However, this has issues with overflow and with maintaining the counter across a distributed system with multiple machines. There are also hybrid approaches that use a combination of both methods.

2 Basic Timestamp Ordering (BASIC T/O)

The basic timestamp ordering protocol (BASIC T/O) allows reads and writes on database objects without using locks. Instead, every database object X is tagged with timestamp of the last transaction that successfully performed a read (denoted as R-TS(X)) or write (denoted as W-TS(X)) on that object. The DBMS then checks these timestamps for every operation. If a transaction tries to access an object in a way which violates the timestamp ordering, the transaction is aborted and restarted. The underlying assumption is that violations will be rare and thus these restarts will also be rare.

Read Operations

For read operations, if TS(T_i) < W-TS(X), this violates timestamp order of T_i with regard to the previous writer of X. Thus, T_i is aborted and restarted with a new timestamp. Otherwise, the read is valid and T_i is allowed to read X. The DBMS then updates R-TS(X) to be the max of R-TS(X) and TS(T_i). It also has to make a local copy of X to ensure repeatable reads for T_i.

Write Operations

For write operations, if TS(T_i) < R-TS(X) or TS(T_i) < W-TS(X), T_i must be restarted. Otherwise, the DBMS allows T_i to write X and updates W-TS(X). Again, it needs to make a local copy of X to ensure repeatable reads for T_i.

Optimization: Thomas Write Rule

An optimization for writes is if TS(T_i) < W-TS(X), the DBMS can instead ignore the write and allow the

transaction to continue instead of aborting and restarting it. This is called the *Thomas Write Rule*. Note that this violates timestamp order of T_i but this is okay because no other transaction will ever read T_i's write to object X.

The Basic T/O protocol generates a schedule that is conflict serializable if it does not use Thomas Write Rule. It cannot have deadlocks because no transaction ever waits. However, there is a possibility of starvation for long transactions if short transactions keep causing conflicts.

It also permits schedules that are not recoverable. A schedule is *recoverable* if transactions commit only after all transactions whose changes they read, commit. Otherwise, the DBMS cannot guarantee that transactions read data that will be restored after recovering from a crash.

Potential Issues:

- High overhead from copying data to transaction's workspace and from updating timestamps.
- Long running transactions can get starved. The likelihood that a transaction will read something from a newer transaction increases.
- Suffers from the timestamp allocation bottleneck on highly concurrent systems.

3 Optimistic Concurrency Control (OCC)

Optimistic concurrency control (OCC) is another optimistic concurrency control protocol which also uses timestamps to validate transactions. OCC works best when the number of conflicts is low. This is when either all of the transactions are read-only or when transactions access disjoint subsets of data. If the database is large and the workload is not skewed, then there is a low probability of conflict, making OCC a good choice.

In OCC, the DBMS creates a *private workspace* for each transaction. All modifications of the transaction are applied to this workspace. Any object read is copied into workspace and any object written is copied to the workspace and modified there. No other transaction can read the changes made by another transaction in its private workspace.

When a transaction commits, the DBMS compares the transaction's workspace *write set* to see whether it conflicts with other transactions. If there are no conflicts, the write set is installed into the "global" database.

OCC consists of three phases:

1. **Read Phase:** Here, the DBMS tracks the read/write sets of transactions and stores their writes in a private workspace.
2. **Validation Phase:** When a transaction commits, the DBMS checks whether it conflicts with other transactions.
3. **Write Phase:** If validation succeeds, the DBMS applies the private workspace changes to the database. Otherwise, it aborts and restarts the transaction.

Validation Phase

The DBMS assigns transactions timestamps when they enter the validation phase. To ensure only serializable schedules are permitted, the DBMS checks T_i against other transactions for RW and WW conflicts and makes sure that all conflicts go one way (from older transactions to younger transactions). The DBMS checks the timestamp ordering of the committing transaction with all other running transactions. Transactions that have not yet entered the validation phase are assigned a timestamp of ∞.

If TS(T_i) $<$ TS(T_j), then one of the following three conditions must hold:

1. T_i completes all three phases before T_j begins

2. T_i completes before T_j starts its Write phase, and T_i does not write to any object read by T_j.
3. T_i completes its Read phase before T_j completes its Read phase, and T_i does not write to any object that is either read or written by T_j.

Potential Issues:

- High overhead for copying data locally into the transaction's private workspace.
- Validation/Write phase bottlenecks.
- Aborts are potentially more wasteful than in other protocols because they only occur after a transaction has already executed.
- Suffers from timestamp allocation bottleneck.

4 Isolation Levels

Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism and limit performance. We may want to use a weaker level of consistency to improve scalability.

Isolation levels control the extent that a transaction is exposed to the actions of other concurrent transactions.

Anomalies:

- **Dirty Read:** Reading uncommitted data.
- **Unrepeatable Reads:** Redoing a read results in a different result.
- **Phantom Reads:** Insertion or deletions result in different results for the same range scan queries.

Isolation Levels (Strongest to Weakest):

1. `SERIALIZABLE`: No Phantoms, all reads repeatable, and no dirty reads.
2. `REPEATABLE READS`: Phantoms may happen.
3. `READ-COMMITTED`: Phantoms and unrepeatable reads may happen.
4. `READ-UNCOMMITTED`: All anomalies may happen.

The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS. There are two additional isolation levels:

1. `CURSOR STABILITY`
   - Between repeatable reads and read committed
   - Prevents Lost Update Anomaly.
   - Default isolation level in **IBM DB2**.
2. `SNAPSHOT ISOLATION`
   - Guarantees that all reads made in a transaction see a consistent snapshot of the database that existed at the time the transaction started.
   - A transaction will commit only if its writes do not conflict with any concurrent updates made since that snapshot.
   - Susceptible to write skew anomaly.

Lecture #18: Multi-Version Concurrency Control

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) is a larger concept than just a concurrency control protocol. It involves all aspects of the DBMS's design and implementation. MVCC is the most widely used scheme in DBMSs. It is now used in almost every new DBMS implemented in last 10 years. Even some systems (e.g., NoSQL) that do not support multi-statement transactions use it.

With MVCC, the DBMS maintains multiple physical versions of a single logical object in the database. When a transaction writes to an object, the DBMS creates a new version of that object. When a transaction reads an object, it reads the newest version that existed when the transaction started.

The fundamental concept/benefit of MVCC is that writers do not block writers and readers do not block readers. This means that one transaction can modify an object while other transactions read old versions.

One advantage of using MVCC is that read-only transactions can read a consistent **snapshot** of the database without using locks of any kind. Additionally, multi-versioned DBMSs can easily support *time-travel queries*, which are queries based on the state of the database at some other point in time (e.g. performing a query on the database as it was 3 hours ago).

There are four important MVCC design decisions:

1. Concurrency Control Protocol
2. Version Storage
3. Garbage Collection
4. Index Management

The choice of concurrency protocol is between the approaches discussed in previous lectures (two-phase locking, timestamp ordering, optimistic concurrency control).

2 Version Storage

This how the DBMS will store the different physical versions of a logical object and how transactions find the newest version visible to them.

The DBMS uses the tuple's pointer field to create a **version chain** per logical tuple, which is essentially a linked list of versions sorted by timestamp. This allows the DBMS to find the version that is visible to a particular transaction at runtime. Indexes always point to the "head" of the chain, which is either the newest or oldest version depending on implementation. A thread traverses chain until it finds the correct version. Different storage schemes determine where/what to store for each version.

Approach #1: Append-Only Storage

All physical versions of a logical tuple are stored in the same table space. Versions are mixed together in the table and each update just appends a new version of the tuple into the table and updates the version chain. The chain can either be sorted *oldest-to-newest* (O2N) which requires chain traversal on look-ups, or *newest-to-oldest* (N2O), which requires updating index pointers for every new version.

Approach #2: Time-Travel Storage

The DBMS maintains a separate table called the time-travel table which stores older versions of tuples. On every update, the DBMS copies the old version of the tuple to the time-travel table and overwrites the tuple in the main table with the new data. Pointers of tuples in the main table point to past versions in the time-travel table.

Approach #3: Delta Storage

Like time-travel storage, but instead of the entire past tuples, the DBMS only stores the deltas, or changes between tuples in what is known as the delta storage segment. Transactions can then recreate older versions by iterating through the deltas. This results in faster writes than time-travel storage but slower reads.

3 Garbage Collection

The DBMS needs to remove *reclaimable* physical versions from the database over time. A version is reclaimable if no active transaction can "see" that version or if it was created by a transaction that was aborted.

Approach #1: Tuple-level GC

With tuple-level garbage collection, the DBMS finds old versions by examining tuples directly. There are two approaches to achieve this:

- **Background Vacuuming**: Separate threads periodically scan the table and look for reclaimable versions. This works with any version storage scheme. A simple optimization is to maintain a "dirty page bitmap," which keeps track of which pages have been modified since the last scan. This allows the threads to skip pages which have not changed.
- **Cooperative Cleaning**: Worker threads identify reclaimable versions as they traverse version chain. This only works with O2N chains.

Approach #2: Transaction-level GC

Under transaction-level garbage collection, each transaction is responsible for keeping track of their own old versions so the DBMS does not have to scan tuples. Each transaction maintains its own read/write set. When a transaction completes, the garbage collector can use that to identify which tuples to reclaim. The DBMS determines when all versions created by a finished transaction are no longer visible.

4 Index Management

All primary key (pkey) indexes always point to version chain head. How often the DBMS has to update the pkey index depends on whether the system creates new versions when a tuple is updated. If a transaction updates a pkey attribute(s), then this is treated as a `DELETE` followed by an `INSERT`.

Managing secondary indexes is more complicated. There are two approaches to handling them.

Approach #1: Logical Pointers

The DBMS uses a fixed identifier per tuple that does not change. This requires an extra indirection layer that maps the logical id to the physical location of the tuple. Then, updates to tuples can just update the mapping in the indirection layer.

Approach #2: Physical Pointers

The DBMS uses the physical address to the version chain head. This requires updating every index when the version chain head is updated.

Lecture #19: Logging Schemes

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Crash Recovery

Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures. When a crash occurs, all the data in memory that has not been committed to disk is at risk of being lost. Recovery algorithms act to prevent loss of information after a crash.

Every recovery algorithm has two parts:

- Actions during normal transaction processing to ensure that the DBMS can recover from a failure.
- Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability.

The key primitives that used in recovery algorithms are UNDO and REDO. Not all algorithms use both primitives.

- **UNDO**: The process of removing the effects of an incomplete or aborted transaction.
- **REDO**: The process of re-instating the effects of a committed transaction for durability.

2 Storage Types

- **Volatile Storage**
  - Data does not persist after power is lost or program exits.
  - Examples: DRAM, SRAM,.
- **Non-Volatile Storage**
  - Data persists after losing power or program exists.
  - Examples: HDD, SDD.
- **Stable Storage**
  - A non-existent form of non-volatile storage that survives all possible failures scenarios.
  - Use multiple storage devices to approximate.

3 Failure Classification

Because the DBMS is divided into different components based on the underlying storage device, there are a number of different types of failures that the DBMS needs to handle. Some of these failures are recoverable while others are not.

Type #1: Transaction Failures

Transactions failures occur when a transaction reaches an error and must be aborted. Two types of errors that can cause transaction failures are logical errors and internal state errors.

- **Logical Errors:** A transaction cannot complete due to some internal error condition (e.g., integrity, constraint violation).

- **Internal State Errors:** The DBMS must terminate an active transaction due to an error condition (e.g., deadlock)

Type #2: System Failures

System failures are unintented failures in hardware or software that must also be accounted for in crash recovery protocols.

- **Software Failure:** There is a problem with the DBMS implementation (e.g., uncaught divide-by-zero exception) and the system has to halt.
- **Hardware Failure:** The computer hosting the DBMS crashes (e.g., power plug gets pulled). We assume that non-volatile storage contents are not corrupted by system crash.

Type #3: Storage Media Failure

Storage media failures are non-repairable failures that occur when the physical storage machine is damaged. When the storage media fails, the DBMS must be restored from an archived version.

- **Non-Repairable Hardware Failure:** A head crash or similar disk failure destroys all or parts of non-volatile storage. Destruction is assumed to be detectable.

4 Buffer Pool Management Policies

The DBMS needs to ensure the following guarantees:

- The changes for any transaction are durable once the DBMS has told somebody that it committed.
- No partial changes are durable if the transaction aborted.

A *steal policy* dictates whether the DBMS allows an uncommitted transaction to overwrite the most recent committed value of an object in non-volatile storage (can a transaction write uncommitted changes to disk).

- **STEAL**: Is allowed
- **NO-STEAL**: Is not allowed.

A *force policy* dictates whether the DBMS requires that all updates made by a transaction are reflected on non-volatile storage before the transaction is allowed to commit.

- **FORCE**: Is required
- **NO-FORCE**: Is not required

Force writes make it easier to recover since all of the changes are preserved but result in poor runtime performance.

The easiest buffer pool management policy to implement is called *NO-STEAL + FORCE*. In the NO-STEAL + FORCE policy, the DBMS never has to undo changes of an aborted transaction because the changes were not written to disk. It also never has to redo changes of a committed transaction because all the changes are guaranteed to be written to disk at commit time. An example of NO-STEAL + FORCE is show in Figure 1.

A limitation of NO STEAL + FORCE is that all of the data that a transaction needs to modify must fit on memory. Otherwise, that transaction cannot execute because the DBMS is not allowed to write out dirty pages to disk before the transaction commits.

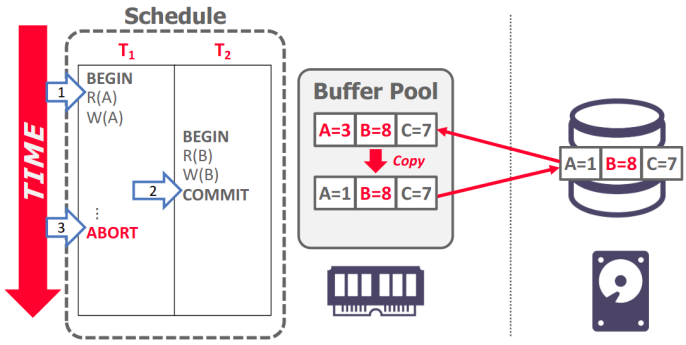


Figure 1: NO-STEAL + FORCE Example – The DBMS is using the NO-STEAL + FORCE buffer policies. All changes from a transaction are only written to disk when the transaction is committed. Once the schedule begins at Step #1, changes from T_1 and T_2 are written to the buffer pool. Because of the FORCE policy, when T_2 commits at Step #2, all of its changes must be written to disk. To do this, the DBMS makes a copy of the memory in disk, applies only the changes from T_2, and writes it back to disk. This is because NO-STEAL prevents the uncommitted changes from T_1 to be written to disk. At Step #3, it is trivial for the DBMS to rollback T_1 since no dirty changes from T_1 are on disk.

5 Shadow Paging

The DBMS maintains two separate copies of the database:

- *master*: Contains only changes from committed txns.
- *shadow*: Temporary database with changes made from uncommitted transactions.

Updates are only made in the shadow copy. When a transaction commits, the shadow is atomically switched to become the new master. This is an example of a NO-STEAL + FORCE system. A high-level example of shadow paging is shown in Figure 2.

Implementation

The DBMS organizes the database pages in a tree structure where the root is a single disk page. There are two copies of the tree, the *master* and *shadow*. The root always points to the current master copy. When a transaction executes, it only makes changes to the shadow copy.

When a transaction wants to commit, the DBMS must install its updates. To do this, it only has overwrite the root to make it points to the shadow copy of the database, thereby swapping the master and shadow. Before overwriting the root, none of the transactions updates are part of the disk-resident database. After overwriting the root, all of the transactions updates are part of the disk resident database.

Recovery

- **Undo**: Remove the shadow pages. Leave the master and DB root pointer alone.

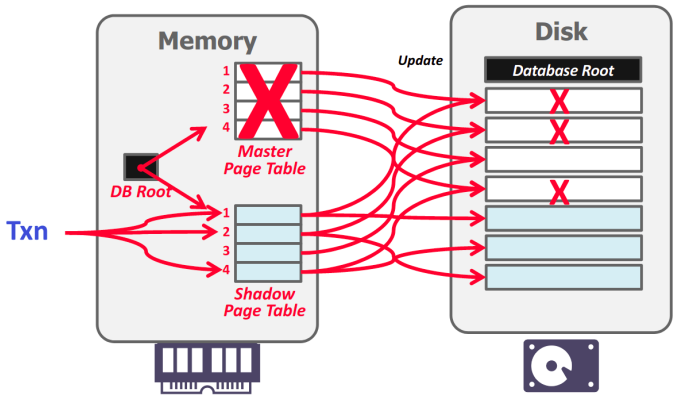


Figure 2: Shadow Paging – The database root points to a master page table which points to the pages on disk. When an updating transaction occurs, a shadow page table is created that points to the same pages as the master. Modifications are made to a temporary space on disk and the shadow table is updated. To commit, the database root pointer is redirected to the shadow table, which becomes the new master.

- **Redo**: Not needed at all.

Disadvantages

A disadvantage of shadow paging is that copying the entire page table is expensive. In reality, only paths in the tree that lead to updated leaf nodes need to be copied, not the entire tree. In addition, the commit overhead of shadow paging is high. Commits require every updated page, page table, and root to be flushed. This causes fragmented data and also requires garbage collection. Another issue is that this only supports one writer transaction at a time or transactions in a batch.

6 Journal File

When a transaction modifies a page, the DBMS copies the original page to a separate journal file before overwriting the master version. After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommited transactions.

7 Write-Ahead Logging

With *write-ahead logging*, the DBMS records all the changes made to the database in a log file (on stable storage) before the change is made to a disk page. The log contains sufficient information to perform the necessary undo and redo actions to restore the database after a crash. The DBMS must write to disk the log file records that correspond to changes made to a database object before it can flush that object to disk. An example of WAL is shown in Figure 3. WAL is an example of a STEAL + NO-FORCE system.

In shadow paging, the DBMS was required to perform writes to random non-contiguous pages on disk. Write-ahead logging allows the DBMS to convert random writes into sequential writes to optimize performance. Thus, almost every DBMS uses write-ahead logging (WAL) because it has the fastest runtime

performance. But the DBMS's recovery time with WAL is slower than shadow paging because it has to replay the log.

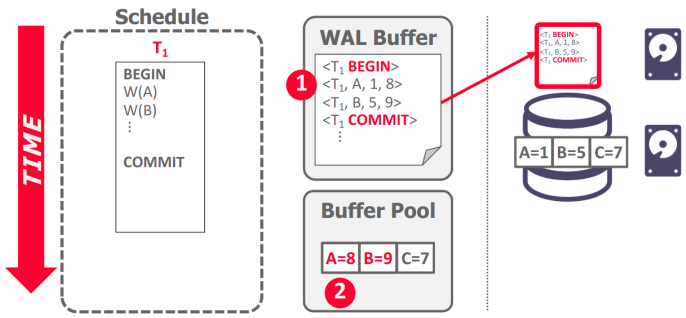


Figure 3: Write Ahead Logging – When the transaction begins, all changes are recorded in the WAL buffer in memory before being made to the buffer pool. When it comes time to commit, the WAL buffer is flushed out to disk. The transaction result can be written once the WAL buffer is safely on disk.

Implementation

The DBMS first stages all of a transaction's log records in volatile storage. All log records pertaining to an updated page are then written to non-volatile storage before the page itself is allowed to be overwritten in non-volatile storage. A transaction is not considered committed until all its log records have been written to stable storage.

When the transaction starts, write a `<BEGIN>` record to the log for each transaction to mark its starting point.

When a transaction finishes, write a `<COMMIT>` record to the log and make sure all log records are flushed before it returns an acknowledgment to the application.

Each log entry contains information about the change to a single object:

- Transaction ID.
- Object ID.
- Before Value (used for UNDO).
- After Value (used for REDO).

The DBMS must flush all of a transaction's log entries to disk before it can tell the outside world that a transaction has successfully committed. The system can use the "group commit" optimization to batch multiple log flushes together to amortize overhead. The DBMS can write dirty pages to disk whenever it wants as long as it's after flushing the corresponding log records.

8 Logging Schemes

The contents of a log record can vary based on the implementation.

Physical Logging:

- Record the byte-level changes made to a specific location in the database.
- Example: Position of a record in a page

Logical Logging:

- Record the high level operations executed by transactions.
- Not necessarily restricted to a single page.
- Requires less data written in each log record than physical logging because each record can update multiple tuples over multiple pages. However, it is difficult to implement recovery with logical logging when there are concurrent transactions in a non-deterministic concurrency control scheme. Additionally recovery takes longer because you must re-execute every transaction.
- Example: The `UPDATE`, `DELETE`, and `INSERT` queries invoked by a transaction.

Physiological Logging:

- Hybrid approach where log records target a single page but do not specify data organization of the page. That is, identify tuples based on a slot number in the page without specifying exactly where in the page the change is located. Therefore the DBMS can reorganize pages after a log record has been written to disk.
- Most common approach used in DBMSs.

Lecture #21: Database Crash Recovery

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Crash Recovery

The DBMS relies on its recovery algorithms to ensure database consistency, transaction atomicity, and durability despite failures. Each recovery algorithm is comprised of two parts:

- Actions during normal transaction processing to ensure ha the DBMS can recover from a failure
- Actions after a failure to recover the database to a state that ensures the atomicity, consistency, and durability of transactions.

Algorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics (ARIES) is a recovery algorithm developed at IBM research in early 1990s for the DB2 system.

There are three key concepts in the ARIES recovery protocol:

- **Write Ahead Logging:** Any change is recorded in log on stable storage before the database change is written to disk (STEAL + NO-FORCE).
- **Repeating History During Redo:** On restart, retrace actions and restore database to exact state before crash.
- **Logging Changes During Undo:** Record undo actions to log to ensure action is not repeated in the event of repeated failures.

2 WAL Records

Write-ahead log records extend the DBMS's log record format to include a globally unique *log sequence number* (LSN). A high level diagram of how log records with LSN's are written is shown in Figure 1.

All log records have an LSN. The `pageLSN` is updated every time a transaction modifies a record in the page. The `flushedLSN` in memory is updated every time the DBMS writes out the WAL buffer to disk.

Various components in the system keep track of **LSNs** that pertain to them. A table of these LSNs is shown in Figure 2.

Each data page contains a `pageLSN`, which is the LSN of the most recent update to that page. The DBMS also keeps track of the max *LSN* flushed so far (`flushedLSN`). Before the DBMS can write page i to disk, it must flush log at least to the point where $\texttt{pageLSN}_i \leq \texttt{flushedLSN}$

3 Normal Execution

Every transaction invokes a sequence of reads and writes, followed by a commit or abort. It is this sequence of events that recovery algorithms must have.

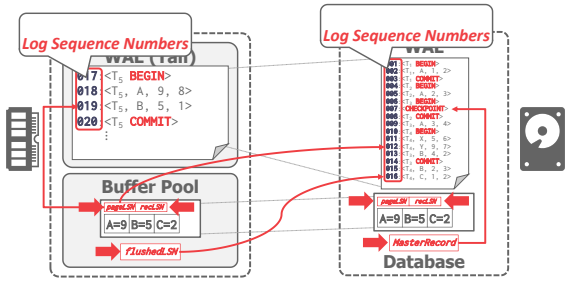


Figure 1: Writing Log Records – Each WAL has a counter of LSNs that is incremented at every step. The page also keeps a pageLSN and a recLSN, which stores the first log record that made the page dirty. The flushedLSN is a pointer to the last LSN that was written out to disk. The MasterRecord points to the last successful checkpoint passed.

Name	Where	Definition
flushedLSN	Memory	Last LSN in log on disk
pageLSN	$page_x$	Newest update to $page_x$
recLSN	$page_x$	Oldest update to $page_x$ since it was last flushed
lastLSN	T_i	Latest record of txn T_i
MasterRecord	Disk	LSN of latest checkpoint

Figure 2: LSN Types – Different parts of the system also maintain different types LSN's that store relevant information.

Transaction Commit

When a transaction goes to commit, the DBMS first writes `COMMIT` record to log buffer in memory. Then the DBMS flushes all log records up to and including the transaction's `COMMIT` record to disk. Note that these log flushes are sequential, synchronous writes to disk. There can be multiple log records per log page. A diagram of a transaction commit is shown in Figure 3.

Once the `COMMIT` record is safely stored on disk, the DBMS returns an acknowledgment back to the application that the transaction has committed. At some later point, the DBMS will write a special `TXN-END` record to log. This indicates that the transaction is completely finished in the system and there will not be anymore log records for it. These `TXN-END` records are used for internal bookkeeping and do not need to be flushed immediately.

Transaction Abort

Aborting a transaction is a special case of the ARIES undo operation applied to only one transaction.

An additional field is added to the log records called the `prevLSN`. This corresponds to the previous LSN for the transaction. The DBMS uses these `prevLSN` values to maintain a linked-list for each transaction that makes it easier to walk through the log to find its records.

A new type of record called the *compensation log record* (CLR) is also introduced. A CLR describes the

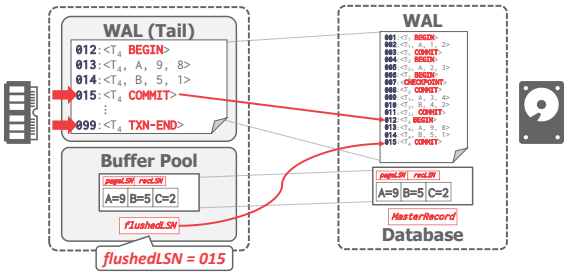


Figure 3: Transaction Commit – After the transaction commits (015), the log is flushed out and the `flushedLSN` is modified to point to the last log record generated. At some later point, a transaction end message is written to signify in the log that this transaction will not appear again.

actions taken to undo the actions of a previous update record. It has all the fields of an update log record plus the *undoNext* pointer (i.e., the next-to-be-undone LSN). The DBMS adds CLRs to the log like any other record but they never need to be undone.

To abort a transaction, the DBMS first appends a `ABORT` record to the log buffer in memory. It then undoes the transaction's updates in reverse order to remove their effects from the database. For each undone update, the DBMS creates **CLR** entry in the log and restore old value. After all of the aborted transaction's updates are reversed, the DBMS then writes a `TXN-END` log record. A diagram of this is shown in Figure 4.

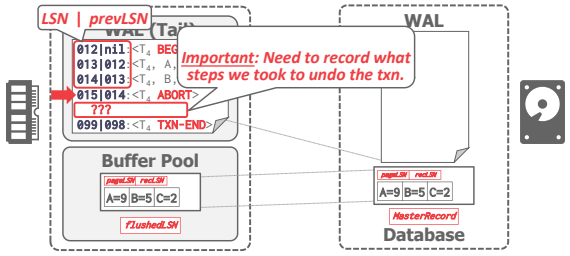


Figure 4: Transaction Abort – The DBMS maintains an LSN and `prevLSN` for each log record that the transaction creates. When the transaction aborts, all of the previous changes are reversed. After the log entries of the reversed changes make it to disk, the DBMS appends the `TXN-END` record to the log for the aborted transaction.

4 Checkpointing

The DBMS periodically takes *checkpoints* where it writes the dirty pages in its buffer pool out to disk. This is used to minimize how much of the log it has to replay upon recovery.

The first two blocking checkpoint methods discussed below pause transactions during the checkpoint pro-

cess. This pausing is necessary to ensure that the DBMS does not miss updates to pages during the checkpoint. Then, a better approach that allows transactions to continue to execute during the checkpoint but requires the DBMS to record additional information to determine what updates it may have missed is presented.

Blocking Checkpoints

The DBMS halts the execution of transactions and queries when it takes a checkpoint to ensure that it writes a consistent snapshot of the database to disk. The is the same approach discussed in previous lecture:

- Halt the start of any new transactions.
- Wait until all active transactions finish executing.
- Flush dirty pages to disk.

Slightly Better Blocking Checkpoints

Like previous checkpoint scheme except that you the DBMS does not have to wait for active transactions to finish executing. The DBMS now records the internal system state as of the beginning of the checkpoint.

- Halt the start of any new transactions.
- Pause transactions while the DBMS takes the checkpoint.

Active Transaction Table (ATT): The ATT represents the state of transactions that are actively running in the DBMS. A transaction's entry is removed after the DBMS completes the commit/abort process for that transaction. For each transaction entry, the ATT contains the following information:

- `transactionId`: Unique transaction identifier
- `status`: The current "mode" of the transaction (Running, Committing, Undo Candidate).
- `lastLSN`: Most recent LSN written by transaction

Note that the ATT contains every transcation without the `TXN-END` log record. This includes both transactions that are either committing or aborting.

Dirty Page Table (DPT): The DPT contains information about the pages in the buffer pool that were modified by uncommitted transactions. There is one entry per dirty page containing the `recLSN` (i.e., the LSN of the log record that first caused the page to be dirty).

The DPT contains all pages that are dirty in the buffer pool. It doesn't matter if the changes were caused by a transaction that is running, committed, or aborted.

Overall, the ATT and the DPT serve to help the DBMS recover the state of the database before the crash via the ARIES recovery protocol.

Fuzzy Checkpoints

A *fuzzy checkpoint* is where the DBMS allows other transactions to continue to run. This is what ARIES uses in its protocol.

The DBMS uses additional log records to track checkpoint boundaries:

- `<CHECKPOINT-BEGIN>`: Indicates the start of the checkpoint. At this point, the DBMS takes a snapshot of the current ATT and DPT, which are referenced in the `<CHECKPOINT-END>` record.
- `<CHECKPOINT-END>`: When the checkpoint has completed. It contains the ATT + DPT, captured just as the `<CHECKPOINT-BEGIN>` log record is written.

5 ARIES Recovery

The ARIES protocol is comprised of three phases. Upon start-up after a crash, the DBMS will execute the following phases as shown in Figure 5:

1. **Analysis**: Read the WAL to identify dirty pages in the buffer pool and active transactions at the time of the crash. At the end of the analysis phase the *ATT* tells the DBMS which transactions were active at the time of the crash. The *DPT* tells the DBMS which dirty pages might not have made it to disk.
2. **Redo**: Repeat all actions starting from an appropriate point in the log.
3. **Undo**: Reverse the actions of transactions that did not commit before the crash.

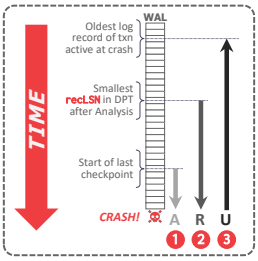


Figure 5: ARIES Recovery: The DBMS starts the recovery process by examining the log starting from the last `BEGIN-CHECKPOINT` found via `MasterRecord`. It then begins the Analysis phase by scanning forward through time to build out ATT and DPT. In the Redo phase, the algorithm jumps to the smallest recLSN, which is the oldest log record that may have modified a page not written to disk. The DBMS then applies all changes from the smallest recLSN. The Undo phase starts at the oldest log record of a transaction active at crash and reverses all changes up to that point.

Analysis Phase

Start from last checkpoint found via the database's `MasterRecord` *LSN*.

1. Scan log forward from the checkpoint.
2. If the DBMS finds a `TXN-END` record, remove its transaction from ATT.
3. All other records, add transaction to ATT with status **UNDO**, and on commit, change transaction status to **COMMIT**.
4. For `UPDATE` log records, if page P is not in the DPT, then add P to DPT and set P's `recLSN` to the log record's *LSN*.

Redo Phase

The goal of this phase is for the DBMS to repeat history to reconstruct its state up to the moment of the crash. It will reapply all updates (even aborted transactions) and redo **CLRs**.

The DBMS scans forward from log record containing smallest `recLSN` in the DPT. For each update log record or CLR with a given *LSN*, the DBMS re-applies the update unless:

- Affected page is not in the DPT, or
- Affected page is in DPT but that record's *LSN* is less than the `recLSN` of the page in DPT, or
- Affected `pageLSN` (on disk) $\geq$ *LSN*.

To redo an action, the DBMS re-applies the change in the log record and then sets the affected page's *pageLSN* to that log record's *LSN*.

At the end of the redo phase, write `TXN-END` log records for all transactions with status COMMIT and remove them from the ATT.

Undo Phase

In the last phase, the DBMS reverses all transactions that were active at the time of crash. These are all transactions with UNDO status in the ATT after the Analysis phase.

The DBMS processes transactions in reverse *LSN* order using the `lastLSN` to speed up traversal. As it reverses the updates of a transaction, the DBMS writes a CLR entry to the log for each modification.

Once the last transaction has been successfully aborted, the DBMS flushes out the log and then is ready to start processing new transactions.

Lecture #22: Introduction to Distributed Databases

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Distributed DBMSs

A distributed DBMS divides a single logical database across multiple physical resources. The application is (usually) unaware that data is split across separated hardware. The system relies on the techniques and algorithms from single-node DBMSs to support transaction processing and query execution in a distributed environment. An important goal in designing a distributed DBMS is fault tolerance (i.e., avoiding a single one node failure taking down the entire system).

Differences between **parallel** and **distributed** DBMSs:

Parallel Database:

- Nodes are physically close to each other.
- Nodes are connected via high-speed LAN (fast, reliable communication fabric).
- The communication cost between nodes is assumed to be small. As such, one does not need to worry about nodes crashing or packets getting dropped when designing internal protocols.

Distributed Database:

- Nodes can be far from each other.
- Nodes are potentially connected via a public network, which can be slow and unreliable.
- The communication cost and connection problems cannot be ignored (i.e., nodes can crash, and packets can get dropped).

2 System Architectures

A DBMS's system architecture specifies what shared resources are directly accessible to CPUs. It affects how CPUs coordinate with each other and where they retrieve and store objects in the database.

A single-node DBMS uses what is called a *shared everything* architecture. This single node executes workers on a local CPU(s) with its own local memory address space and disk.

Shared Memory

An alternative to shared everything architecture in distributed systems is *shared memory*. CPUs have access to common memory address space via a fast interconnect. CPUs also share the same disk.

In practice, most DBMSs do not use this architecture, as it is provided at the OS / kernel level. It also causes problems, since each process's scope of memory is the same memory address space, which can be modified by multiple processes.

Each processor has a global view of all the in-memory data structures. Each DBMS instance on a processor has to "know" about the other instances.

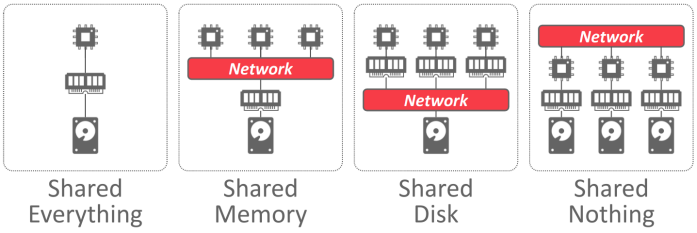


Figure 1: Database System Architectures – Four system architecture approaches ranging from sharing everything (used by non distributed systems) to sharing memory, disk, or nothing.

Shared Disk

In a *shared disk* architecture, all CPUs can read and write to a single logical disk directly via an interconnect, but each have their own private memories. This approach is more common in cloud-based DBMSs.

The DBMS's execution layer can scale independently from the storage layer. Adding new storage nodes or execution nodes does not affect the layout or location of data in the other layer.

Nodes must send messages between them to learn about other node's current state. That is, since memory is local, if data is modified, changes must be communicated to other CPUs in the case that piece of data is in main memory for the other CPUs.

Nodes have their own buffer pool and are considered stateless. A node crash does not affect the state of the database since that is stored separately on the shared disk. The storage layer persists the state in the case of crashes.

Shared Nothing

In a *shared nothing* environment, each node has its own CPU, memory, and disk. Nodes only communicate with each other via network.

It is more difficult to increase capacity in this architecture because the DBMS has to physically move data to new nodes. It is also difficult to ensure consistency across all nodes in the DBMS, since the nodes must coordinate with each other on the state of transactions. The advantage, however, is that shared nothing DBMSs can potentially achieve better performance and are more efficient then other types of distributed DBMS architectures.

3 Design Issues

Distributed DBMSs aim to maintain *data transparency*, meaning that users should not be required to know where data is physically located, or how tables are partitioned or replicated. The details of how data is being

stored is hidden from the application. In other words, a SQL query that works on a single-node DBMS should work the same on a distributed DBMS.

The key design questions that distributed database systems must address are the following:

- How does the application find data?
- How should queries be executed on a distributed data? Should the query be pushed to where the data is located? Or should the data be pooled into a common location to execute the query?
- How does the DBMS ensure correctness?

Another design decision to make involves deciding how the nodes will interact in their clusters. Two options are *homogeneous* and *heterogeneous* nodes, which are both used in modern-day systems.

Homogeneous Nodes: Every node in the cluster can perform the same set of tasks (albeit on potentially different partitions of data), lending itself well to a shared nothing architecture. This makes provisioning and failover "easier". Failed tasks are assigned to available nodes.

Heterogeneous Nodes: Nodes are assigned specific tasks, so communication must happen between nodes to carry out a given task. Can allow a single physical node to host multiple "virtual" node types for dedicated tasks. Can independently scale from one node to other.

4 Partitioning Schemes

Distributed system must partition the database across multiple resources, including disks, nodes, processors. This process is sometimes called *sharding* in NoSQL systems. When the DBMS receives a query, it first analyzes the data that the query plan needs to access. The DBMS may potentially send fragments of the query plan to different nodes, then combines the results to produce a single answer.

The goal of a partitioning scheme is to maximize single-node transactions, or transactions that only access data contained on one partition. This allows the DBMS to not need to coordinate the behavior of concurrent transactions running on other nodes. On the other hand, a distributed transaction accesses data at one or more partitions. This requires expensive, difficult coordination, discussed in the below section.

For *logically partitioned nodes*, particular nodes are in charge of accessing specific tuples from a shared disk. For *physically partitioned nodes*, each shared nothing node reads and updates tuples it contains on its own local disk.

Implementation

The simplest way to partition tables is *naive data partitioning*. Each node stores one table, assuming enough storage space for a given node. This is easy to implement because a query is just routed to a specific partitioning. This can be bad, since it is not scalable. One partition's resources can be exhausted if that one table is queried on often, not using all nodes available. See Figure 2 for an example.

More commonly used is *horizontal partitioning*, which splits a table's tuples into disjoint subsets. Choose column(s) that divides the database equally in terms of size, load, or usage, called the *partitioning key(s)*. The DBMS can partition a database physically (shared nothing) or logically (shared disk) via hash partitioning or range partitioning. See Figure 3 for an example.

Another common approach is *Consistent Hashing*. Consistent Hashing assigns every node to a location on some logical ring. Then the hash of every partition key maps to some location on the ring. The node that is closest to the key in the clockwise direction is responsible for that key. See Figure 4 for an example. When a node is added or removed, keys are only moved between nodes adjacent to the new/removed node. A replication factor of n means that each key is replicated at the n closest nodes in the clockwise direction.

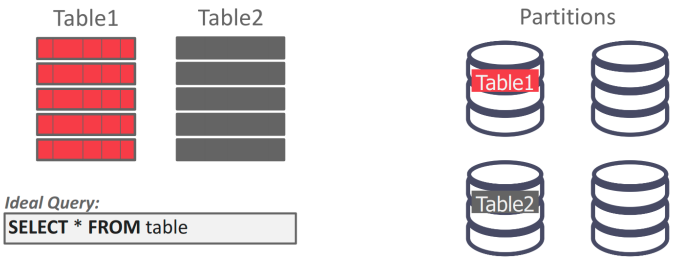


Figure 2: Naive Table Partitioning – Given two tables, place all the tuples in table one into one partition and the tuples in table two into the other.

Logical Partitioning: A node is responsible for a set of keys, but it doesn't actually store those keys. This is commonly used in a shared disk architecture.

Physical Partitioning: A node is responsible for a set of keys, and it physically stores those keys. This is commonly used in a shared nothing architecture.

5 Distributed Concurrency Control

A distributed transaction accesses data at one or more partitions, which requires expensive coordination.

Centralized coordinator

The centralized coordinator acts as a global "traffic cop" that coordinates all the behavior. See Figure 5 for a diagram.

Middleware

Centralized coordinators can be used as *middleware*, which accepts query requests and routes queries to correct partitions.

Decentralized coordinator

In a decentralized approach, nodes organize themselves. The client directly sends queries to one of the partitions. This *home partition* will send results back to the client. The home partition is in charge of communicating with other partitions and committing accordingly.

Centralized approaches give way to a bottleneck in the case that multiple clients are trying to acquire locks on the same partitions. It can be better for distributed 2PL as it has a central view of the locks and can handle deadlocks more quickly. This is non-trivial with decentralized approaches.

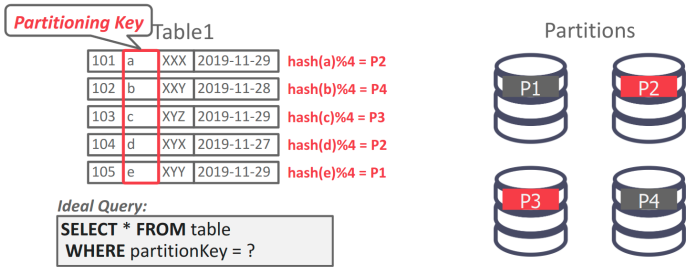


Figure 3: Horizontal Table Partitioning – Use hash partitioning to decide where to send the data. When the DBMS receives a query, it will use the table's partitioning key(s) to find out where the data is.

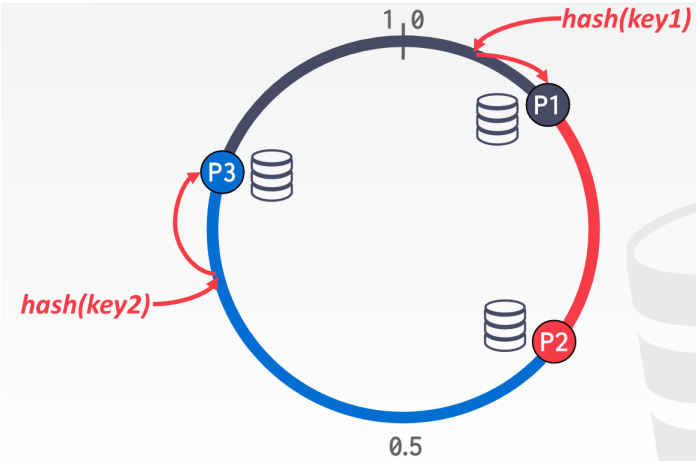


Figure 4: Consistent Hashing – All nodes are responsible for some portion of hash ring. Here node P1 is responsible for storing key1 and node P3 is responsible for storing key2.

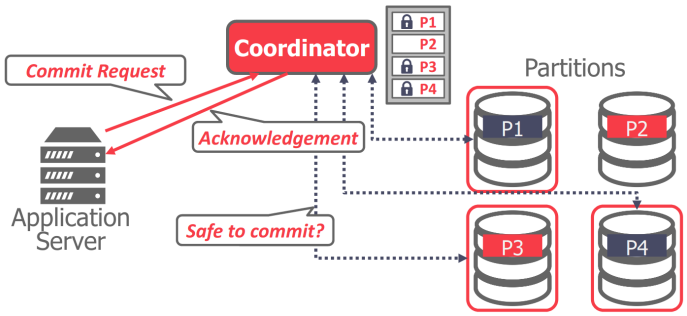


Figure 5: Centralized Coordinator – The client communicates with the coordinator to acquire locks on the partitions that the client wants to access. Once it receives an acknowledgement from the coordinator, the client sends its queries to those partitions. Once all queries for a given transaction are done, the client sends a commit request to the coordinator. The coordinator then communicates with the partitions involved in the transaction to determine whether the transaction is allowed to commit.

Lecture #23: Distributed OLTP Databases

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 OLTP VS. OLAP

On-line Transaction Processing (OLTP)

- Short lived read/write transactions.
- Small footprint.
- Repetitive operations.

On-line Analytical Processing OLAP

- Long-running, read-only queries.
- Complex joins.
- Exploratory queries.

2 Distributed Transactions

A transaction is "distributed" if it accesses data on multiple nodes. Executing these transactions is more challenging than single-node transactions because now when the transaction commits, the DBMS has to make sure that all the nodes agree to commit the transaction. The DBMS ensure that the database provides the same ACID guarantees as a single-node DBMS even in the case of node failures or message loss.

One can assume that all nodes in a distributed DBMS are well-behaved and under the same administrative domain. In other words, given that there is not a node failure, a node which is told to commit a transaction will commit the transaction. If the other nodes in a distributed DBMS cannot be trusted, then the DBMS needs to use a *byzantine fault tolerant* protocol (e.g., blockchain) for transactions.

3 Atomic Commit Protocols

When a multi-node transaction finishes, the DBMS needs to ask all of the nodes involved whether it is safe to commit. Depending on the protocol, a majority of the nodes or all of the nodes may be needed to commit. Examples include:

- Two-Phase Commit (Common)
- Three-Phase Commit (Uncommon)
- Paxos (Common)
- Raft (Common)
- ZAB (**Apache Zookeeper**)
- Viewstamped Replication (first probably correct protocol)

Two-Phase Commit (2PC) blocks if coordinator fails after the prepare message is sent, until the coordinator recovers. Paxos, on the other hand, is non-blocking if a majority participants are alive, provided there is a

sufficiently long period without further failures. 2PC is used often if the nodes are in the same data center because of the number of round trips could be less than for Paxos, assuming that nodes do not fail often and are not malicious.

Two-Phase Commit

The client sends a *Commit Request* to the coordinator. In the first phase of this protocol, the coordinator sends a *Prepare* message, essentially asking the participant nodes if the current transaction is allowed to commit. If a given participant verifies that the given transaction is valid, they send an *OK* to the coordinator. If the coordinator receives an *OK* from all the participants, the system can now go into the second phase in the protocol. If anyone sends an *Abort* to the coordinator, the coordinator sends an *Abort* to the client.

The coordinator sends a *Commit* to all the participants, telling those nodes to commit the transaction, if all the participants sent an *OK*. Once the participants respond with an *OK*, the coordinator can tell the client that the transaction is committed. If the transaction was aborted in the first phase, the participants receive an *Abort* from the coordinator, to which they should respond to with an *OK*. Either everyone commits or no one does. The coordinator can also be a participant in the system.

Additionally, in the case of a crash, all nodes keep track of a non-volatile log of the outcome of each phase. Nodes block until they can figure out the next course of action. If the coordinator crashes, the participants must decide what to do. A safe option is just to abort. Alternatively, the nodes can communicate with each other to see if they can commit without the explicit permission of the coordinator. If a participant crashes, the coordinator assumes that it responded with an abort if it has not sent an acknowledgement yet.

Optimizations:

- *Early Prepare Voting* – If the DBMS sends a query to a remote node that it knows will be the last one executed there, then that node will also return their vote for the prepare phase with the query result.
- *Early Acknowledgement after Prepare* – If all nodes vote to commit a transaction, the coordinator can send the client an acknowledgement that their transaction was successful before the commit phase finishes.

Paxos

Paxos (along with Raft) is more prevalent in modern systems than 2PC. It is a less strict version of 2PC. This is a consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed. This protocol does not block if a majority of participants are available and has probably minimal message delays in the best case. For Paxos, the coordinator is called the **proposer** and participants are called **acceptors**.

The client will send a *Commit Request* to the proposer. The proposer will send a *Propose* to the other nodes in the system, or the acceptors. A given acceptor will send an *Agree* if they have not already sent an *Agree* on a higher logical timestamp. Otherwise, they send a *Reject*.

Once the majority of the acceptors sent an *Agree*, the proposer will send a *Commit*. The proposer must wait to receive an *Accept* from the majority of acceptors before sending the final message to the client saying that the transaction is committed, unlike 2PC.

Use exponential back off times for trying to propose again after a failed proposal, to avoid dueling proposers.

Multi-Paxos: If the system elects a single leader that oversees proposing changes for some period, then it can skip the propose phase. The system periodically renews who the leader is using another Paxos round. When there is a failure, the DBMS can fall back to full Paxos.

4 Replication

The DBMS can replicate data across redundant nodes to increase availability. In other words, if a node goes down, the data is not lost, and the system is still alive and does not need to be rebooted. One can use Paxos to determine which replica to write data to.

Number of Primary Nodes

In **Primary-Replica**, all updates go to a designated primary for each object. The primary propagates updates to its replicas without an atomic commit protocol, coordinating all updates that come to it. Read-only transactions may be allowed to access replicas if the most up-to-date information is not needed. If the primary goes down, then hold an election to select a new primary.

In **Multi-Primary**, transactions can update data objects at any replica. Replicas must synchronize with each other using an atomic commit protocol like Paxos or 2PC.

K-Safety

K-safety is a threshold for determining the fault tolerance of the replicated database. The value *K* represents the number of replicas per data object that must always be available. If the number of replicas goes below this threshold, then the DBMS halts execution and takes itself offline. A higher value of K reduces risk of losing data. It is a threshold to determine how available a system can be.

Propagation Scheme

When a transaction commits on a replicated database, the DBMS decides whether it must wait for that transaction's changes to propagate to other nodes before it can send the acknowledgement to the application client. There are two propagation levels: Synchronous (strong consistency) and asynchronous (eventual consistency).

In a *synchronous* scheme, the primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes. Then, the primary can notify the client that the update has succeeded. It ensures that the DBMS will not lose any data due to strong consistency. This is more common in a traditional DBMS.

In an *asynchronous* scheme, the primary immediately returns the acknowledgement to the client without waiting for replicas to apply the changes. Stale reads can occur in this approach, since updates may not be fully applied to replicas when read is occurring. If some data loss can be tolerated, this option can be a viable optimization. This is used commonly in NoSQL systems.

Propagation Timing

For *continuous* propagation timing, the DBMS sends log messages immediately as it generates them. Note that a commit or abort message needs to also be sent. Most systems use this approach.

For *on commit* propagation timing, the DBMS only sends the log messages for a transaction to the replicas once the transaction is committed. This does not waste time for sending log records for aborted transactions. It does make the assumption that a transaction's log records fit entirely in memory.

Active vs Passive

There are multiple approaches to applying changes to replicas. For *active-active*, a transaction executes at each replica independently. At the end, the DBMS needs to check whether the transaction ends up with the

same result at each replica to see if the replicas committed correctly. This is difficult since now the ordering of the transactions must sync between all the nodes, making it less common.

For *active-passive*, each transaction executes at a single location and propagates the overall changes to the replica. The DBMS can either send out the physical bytes that were changed, which is more common, or the logical SQL queries.

5 CAP Theorem

The *CAP Theorem*, proposed by Eric Brewer and later proved in 2002 at MIT, explained that it is impossible for a distributed system to always be Consistent, Available, and Partition Tolerant. Only two of these three properties can be chosen.

Consistency is synonymous with linearizability for operations on all nodes. Once a write completes, all future reads should return the value of that write applied or a later write applied. Additionally, once a read has been returned, future reads should return that value or the value of a later applied write. NoSQL systems compromise this property in favor of the latter two. Other systems will favor this property and one of the latter two.

Availability is the concept that all up nodes can satisfy all requests.

Partition tolerance means that the system can still operate correctly despite some message loss between nodes that are trying to reach consensus on values. If consistency and partition tolerance is chosen for a system, updates will not be allowed until a majority of nodes are reconnected, typically done in traditional or NewSQL DBMSs.

6 Federated Databases

These are distributed architectures that connect together multiple DBMSs into a single logical system. This is more popular in bigger companies. A query can access data at any location. This is hard due to different data models, query languages, and limitations of each individual DBMS. Additionally, there is no easy way to optimize queries. Lastly, there is a lot of data copying that is involved.

For example, say there is an application server which makes some queries. These queries then go through a middleware layer (which will convert the query into a readable format for a given DBMS used in the bigger system) that via *connectors*, will go through the multiple back-end DBMSs that are deployed in the system. The middleware will then handle the results returned from the DBMSs.

PostgreSQL is in the best position to successfully deploy a federated database using its *foreign data wrappers*. It allows a user to use data from another system within a given Postgres session.

Lecture #24: Distributed OLAP Databases

15-445/645 Database Systems (Fall 2021)

https://15445.courses.cs.cmu.edu/fall2021/

Carnegie Mellon University
Lin Ma

1 Decision Support Systems

For a read-only OLAP database, it is common to have a bifurcated environment, where there are multiple instances of OLTP databases that ingest information from the outside world which is then fed into the back-end OLAP database, sometimes called a *data warehouse*. There is an intermediate step called *ETL*, or **E**xtract, **T**ransform, and **L**oad, which combines the OLTP databases into a universal schema for the data warehouse.

Decision support systems (DSS) are applications that serve the management, operations, and planning levels of an organization to help people make decisions about future issues and problems by analyzing historical data stored in a data warehouse.

The two approaches for modeling an analytical database are *star schemas* and *snowflake schemas*.

Star Schema

Star schemas contain two types of tables: *fact tables* and *dimension tables*. The fact table contains multiple "events" that occur in the application. It will contain the minimal unique information per event, and then the rest of the attributes will be foreign key references to outer dimension tables. The dimension tables contain redundant information that is reused across multiple events. In a star schema, there can only be one dimension-level out from the fact table. Since the data can only have one level of dimension tables, it can have redundant information. Denormalized data models may incur integrity and consistency violations, so replication must be handled accordingly. Queries on star schemas will (usually) be faster than a snowflake schema because there are fewer joins. An example of a star schema is shown in Figure 1.

Snowflake Schema

Snowflake schemas are similar to star schemas except that they allow for more than one dimension out from the fact table. They take up less storage space, but they require more joins to get the data needed for a query. For this reason, queries on star schemas are usually faster. An example of a snowflake schema is shown in Figure 2.

2 Execution Models

A distributed DBMS's execution model specifies how it will communicate between nodes during query execution. Two approaches to executing a query are *pushing* and *pulling*.

Pushing a Query to Data

For the first approach, the DBMS sends the query (or a portion of it) to the node that contains the data. It then performs as much filtering and processing as possible where data resides before transmitting over network. The result is then sent back to where the query is being executed, which uses local data and the data sent to it, to complete the query. This is more common in a shared nothing system.

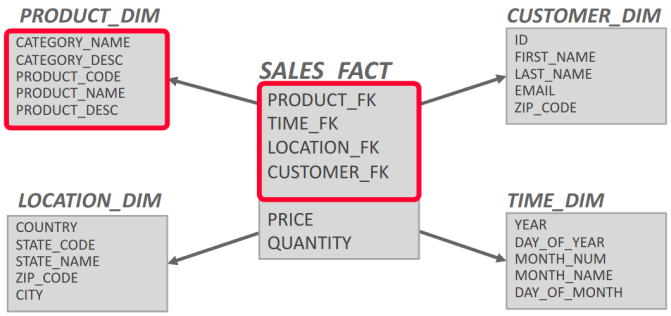


Figure 1: Star Schema – The center of the schema is the SALES fact table that contains key references to outer dimension tables. Because star schemas are only one-dimensional, the outer dimensional tables cannot point to other dimension tables.

Pulling Data to Query

For the second approach, the DBMS brings the data to the node that is executing a query that needs it for processing. In other words, nodes detect which partitions of the data they can do computation on and pull from storage accordingly. Then, the local operations are propagated to one node, which does the operation on all the intermediary results. This is normally what a shared disk system would do. The problem with this is that the size of the data relative to the size of the query could be very different. A filter can also be sent to only retrieve the data needed from disk.

Query Fault Tolerance

The data that a node receives from remote sources are cached in the buffer pool. This allows the DBMS to support intermediate results that are larger than the amount of memory available. Ephemeral pages, however, are not persisted after a restart. Therefore, a distributed DBMS must consider what happens to a long-running OLAP query if a node crashes during execution.

Most shared-nothing distributed OLAP DBMSs are designed to assume that nodes do not fail during query execution. If one node fails during query execution, then the whole query fails, which entails the entire query executing from the start. This can be expensive, as some OLAP queries can take days to execute.

The DBMS could take a snapshot of the intermediate results for a query during execution to allow it to recover if nodes fail. This operation is expensive, however, because writing data to disk is slow.

3 Query Planning

All the optimizations that we talked about before are still applicable in a distributed environment, including predicate pushdown, early projections, and optimal join orderings. Distributed query optimization is even harder because it must consider the physical location of data in the cluster and data movement costs.

One approach is to generate a single global query plan and then distribute *physical operators* to nodes, breaking it up into partition-specific fragments. Most systems implement this approach.

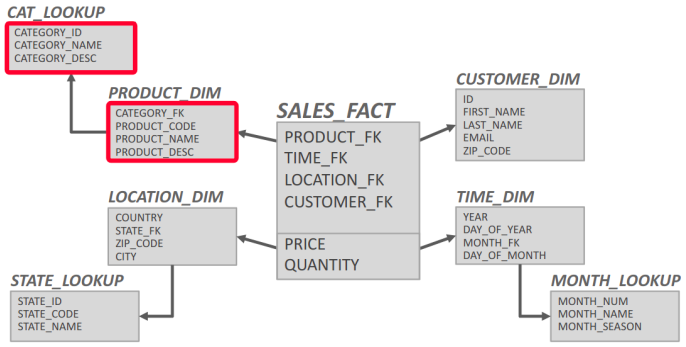


Figure 2: Snowflake Schema – The category information in the product dimension table can be broken out in the snowflake table.

Another approach is to take the *SQL* query and rewrite the original query into partition-specific queries. This allows for local optimization at each node. SingleStore and Vitess are examples of systems that use this approach.

4 Distributed Join Algorithms

For analytical workloads, the majority of the time is spent doing joins and reading from disk, showing the importance of this topic. The efficiency of a distributed join depends on the target tables' partitioning schemes.

One approach is to put entire tables on a single node and then perform the join. However, the DBMS loses the parallelism of a distributed DBMS, which defeats the purpose of having a distributed DBMS. This option also entails costly data transfer over the network.

To join tables R and S, the DBMS needs to get the proper tuples on the same node. Once there, it then executes the same join algorithms discussed earlier in the semester. One should always send the minimal amount needed to compute the join, sometimes entailing entire tuples.

There are four scenarios for distributed join algorithms.

Scenario 1

One of the tables is replicated at every node and the other table is partitioned across nodes. Each node joins its local data in parallel and then sends their results to a coordinating node.

Scenario 2

Both tables are partitioned on the join attribute, with IDs matching on each node. Each node performs the join on local data and then sends to a node for coalescing.

Scenario 3

Both tables are partitioned on different keys. If one of the tables is small, then the DBMS broadcasts that table to all nodes. Local joins are computed and then those joins are sent to a common node to operate the final join. This is known as a *broadcast join*.

Scenario 4

This is the worst case scenario. Both tables are not partitioned on the join key. The DBMS copies the tables by reshuffling them across nodes. Local joins are computed and then the results are sent to a common node for the final join. If there isn't enough disk space, a failure is unavoidable. This is called a *shuffle join*.

Semi-Join

A *semi-join* is a join operator where the result only contains columns from the left table. Distributed DBMSs use semi-join to minimize the amount of data sent during joins.

It is like a natural join, except that the attributes on the right table that are not used to compute the join are restricted.

5 Cloud Systems

Vendors provide *database-as-a-service* (DBaaS) offerings that are managed DBMS environments.

Newer systems are starting to blur the lines between shared-nothing and shared-disk. For example, **Amazon S3** allows for simple filtering before copying data to compute nodes. There are two types of cloud systems, managed or cloud-native DBMSs.

Managed DBMSs

In a managed DBMS, no significant modification to the DBMS to be "aware" that it is running in a cloud environment. It provides a way to abstract away all the backup and recovery for the client. This approach is deployed in most vendors.

Cloud-Native DBMS

A cloud-native system is designed explicitly to run in a cloud environment. This is usually based on a shared-disk architecture. This approach is used in **Snowflake**, **Google BigQuery**, **Amazon Redshift**, and **Microsoft SQL Azure**.

Serverless Databases

Rather than always maintaining compute resources for each customer, a *serverless DBMS* evicts tenants when they become idle, checkpointing the current progress in the system to disk. Now, a user is only paying for storage when not actively querying. A diagram of this is shown in Figure 3.

6 Disaggregated Components

Many existing libraries/systems implement a single component of a distributed database. Distributed databases can then leverage these components instead of re-implementing it themselves. Additionally different distributed databases can share components with each other.

Notable examples are:

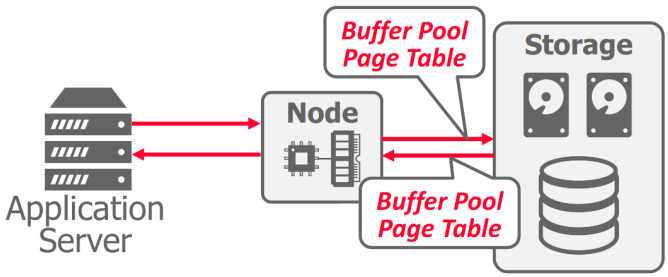


Figure 3: Serverless Database – When the application server becomes idle, the user must pay for resources in the node that are not being used. In a serverless database, when the application server stops, the DBMS takes a snapshot of pages in the buffer pool and writes it out to shared disk so that the computation can be stopped. When the application server returns, the buffer pool page table restores the previous state in the node.

System Catalogs: HCatalog, Google Data Catalog, Amazon Glue Data Catalog,

Node Management: Kubernetes, Apache YARN, Cloud Vendor Tools

Query Optimizers: Greenplum Orca, Apache Calcite

7 Universal Formats

Most DBMSs use a proprietary on-disk binary file format for their databases. The only way to share data between systems is to convert data into a common text-based format, including CSV, JSON, and XML. There are new open-source binary file formats, which cloud vendors and distributed database systems support, that make it easier to access data across systems. Writing a custom file format would give way to better compression and performance, but this gives way to better interoperability.

Notable examples of universal database file formats:

- **Apache Parquet:** Compressed columnar storage from Cloudera/Twitter.
- **Apache ORC:** Compressed columnar storage from **Apache Hive**.
- **Apache CarbonData:** Compressed columnar storage with indexes from Huawei.
- **Apache Iceberg:** Flexible data format that supports schema evolution from Netflix.
- **HDF5:** Multi-dimensional arrays for scientific workloads.
- **Apache Arrow:** In-memory compressed columnar storage from Pandas/Dremio.