Lecture #01: Relational Model & Relational Algebra

15-445/645 Database Systems (Fall 2019)

https://15445.courses.cs.cmu.edu/fall2019/

Carnegie Mellon University

Prof. Andy Pavlo

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 tith an invalid string?
  - How do we store theat there are multiple artists on one album?
- **Implementation**
  - How do you 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 software that allows applications to store and analyze information in a database.

A general-purpose DBMS is designed to allow the definition, creation, querying, updating, 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 is 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:** 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 specifed 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 atttributes.

Syntax: $(R \cup S)$.

Intersection

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

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 atttributes.

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 attrubite 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: Advanced SQL

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Relational Languages

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

- **SQL**: Structured Query Language
- IBM originally called it "SEQUEL".
- Comprised of different classes of commands:
  1. Data Manipulation Language (DML): SELECT, INSERT, UPDATE, DELETE.
  2. Data Definition Language (DDL): Schema definition.
  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 in order to claim they support SQL. Each vendor follows the standard to a certain degree but there are many proprietary extensions.

3 Aggregates

- An aggregation function takes in a bag of tuples as its input and then produces a single scalar value as its output. Can only be used in SELECT output list.
  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:

```
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

```
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)

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

- Thus, other columns outside aggregate must be aggregated or used in a `GROUP BY` command:

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

- `HAVING`: Filters output results after aggregation. Like a `WHERE` clause for a `GROUP BY`

```
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;
```

4 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.

5 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 name of the columns in the output query do not have to match.

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

6 Output Control

- **ORDER BY:** Since results SQL are unordered, you have to use the ORDER BY clause to impose a sort on tuples:

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

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

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

  You 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;
```

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

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

  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 you use an ORDER BY clause with a LIMIT, the tuples in the result could be different on each invocation.

7 Nested Queries

- Invoke queries inside of other queries to execute more complex logic within a single query. The scope of outer query is included in inner query (i.e. inner query can access attributes from outer query), but not the other way around.
- Inner queries can appear (almost) anywhere in 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.
- Nest 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.

8 Window Functions

- Performs “moving” calculation across set of tuples. Like an aggregation but it still returns the original tuples.
- **Functions:** Can be any of the aggregation functions that we discussed above. Can also be a 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;
  ```

  You 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.

9 Common Table Expressions

- Common Table Expressions (CTEs) are an alternative to windows or nested queries to writing more complex queries. One can think of a CTE like a temporary table for just one 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". The query at the bottom then just returns all the attributes from "cteName".

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

- You 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 can 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.
  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 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Storage

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

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 storage devices have larger the capacities but are much slower and farther away from the CPU. 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 need to be provided 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 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) or spinning hard drives (HDD).

There is also a new class of storage devices that are coming out soon called *non-volatile memory*. These devices are designed to be the best of both worlds: almost as fast as DRAM but with the persistence of disk. We will not cover these devices.

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

We will focus on how we can hide the latency of the disk rather than focusing on 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, then 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 the database files is organized into pages, and the first page is the directory page. In order 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 movement back and forth between disk and memory. The DBMS also have 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 the page in memory. The buffer pool manager will ensure that the page is there while the execution engine is operating on that 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, it must be managed carefully. We do not want large stalls from fetching something from disk to slow down everything else. So 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, this will block the process.

- 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 the 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 it is 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 and 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, then when 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 ways to find the location of the page a DBMS wants on the disk, and a heap file organization is one of the 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 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.
- Some systems require pages to be self-contained (e.g oracle).

A strawman approach to laying out data is to keep track of how many tuples the DBMS has stored in a page and then every time it adds a new tuple, it appends the tuple to the end. However, problems arise when it deletes a tuple or when the 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 and the offset of the starting location of 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 the DBMS ends up with write amplification (it re-writs the same data over and over again).

8 Tuple Layout

A tuple is essentially a sequence of bytes. It is 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, but 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 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Data Representation

Tuples' data 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 four main types 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

- Inexact, variable-precision numeric type that uses the "native" C/C++ types specified by IEEE-754 standard. These values are also fixed length.
- Variable-precision numbers are faster to compute than arbitrary precision numbers because the CPU can execute instructions on them directly.
- 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 with additional meta-data that will tell the system things like 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.
- Example: `NUMERIC`, `DECIMAL`.

Variable Length Data

- An array of bytes of arbitrary length.
- Has a header that keeps track of the length of the string to make it easy to jump to the next value.
- Most DBMSs do not allow a tuple to exceed the size of a single page, so they solve this issue by writing the value on an overflow page and have the tuple contain a reference to that page.
- 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 our database is storing photo information, we 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.
- Example: `VARCHAR`, `VARBINARY`, `TEXT`, `BLOB`.

Dates and Times

- Usually, these are represented as the number of (micro/milli)seconds since the unix epoch.
- Example: `TIME`, `DATE`, `TIMESTAMP`.

System Catalogs

In order for the DBMS to be able to read these values, it maintains an internal catalog to tell it meta-data about the databases. The meta-data will contain 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.

2 Workloads

OLTP: On-line Transaction Processing

- Fast, short running operations
- Queries operate on single entity at a time
- More writes than reads
- Repetitive operations
- Usually the kind of application that people build first
- Example: User invocations of Amazon. They can add things to their cart, they can make purchases, but the actions only affect their account.

OLAP: On-line Analyitical Processing

- Long running, more complex queries
- Reads large portions of the database
- Exploratory queries
- Deriving new data from data collected on the OLTP side
- Example: Compute the five most bought items over a one month period for these geographical locations.

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)

The DBMS stores all of the attributes for a single tuple contiguously, so NSM is also known as a "row store." This approach is ideal for OLTP workloads where transactions tend to operate only an individual entity and insert heavy workloads. 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.

There are two different ways to organize a NSM database:

- **Heap-Organized Tables:** Tuples are stored in blocks called a heap, and the heap does not necessarily define an order.
- **Index-Organized Tables:** Tuples are stored in the primary key index itself, but different from a clustered index.

Decomposition Storage Model (DSM)

The DBMS stores a single attribute (column) for all tuples contiguously in a block of data. Also known as a "column store." This model is ideal for OLAP workloads where read-only queries 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 we are using a column store, we can use:

- **Fixed-length offsets**: Start by assuming the attributes are all fixed-length. Then when the system wants the attribute for a specific tuple, it knows how to jump to that spot in the file. To accommodate the variable-length fields, the system can pad them so that they are all the same length, or you could use a dictionary that takes a fixed-size integer and maps the integer to the value.
- **Embedded Tuple Ids**: For every attribute in the columns, store the tuple id with it. The system would also need extra information to tell it how to jump to every attribute that has that id.

Most DBMSs use fixed-length offsets.

Row stores are usually better for OLTP, while column stores ar better for OLAP.

Lecture #05: Buffer Pools

15-445/645 Database Systems (Fall 2019)

`https://15445.courses.cs.cmu.edu/fall2019/`

Carnegie Mellon University
Prof. Andy Pavlo

1 Locks vs. Latches

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

Locks

- Protect the database logical contents (e.g., tuples, tables, databases) from other transactions.
- Held for transaction duration.
- Need to be able to rollback changes.

Latches

- Protects the critical sections of the DBMS's internal data structures from other threads.
- Held for operation duration.
- Do not need to be able to rollback changes.

2 Buffer Pool

The *buffer pool* is an in-memory cache of pages read from disk. The DBMS always knows better so we want to manage memory and pages ourselves

It is a 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 these frames

Meta-data maintained by the buffer pool:

- **Page Table:** 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.
- **Dirty-flag:** Threads set this flag when it modifies a page. This indicates to storage manager that the page must be written back to disk.
- **Pin Counter:** This 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.

Optimizations:

- **Multiple Buffer Pools:** The DBMS can also have multiple buffer pools for different purposes. This helps reduce latch contention and improves locality
- **Pre-Fetching:** The DBMS can also optimize by pre fetching pages based on the query plan. Commonly done when accessing pages sequentially.
- **Scan Sharing:** Query cursors can attach to other cursors and scan pages together.

Allocation Policies:

- **Global Policies:** How a DBMS should make decisions for all active txns.
- **Local Policies:** Allocate frames to a specific txn without considering the behavior of concurrent txns.

3 Replacement Policies

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:

- Correctness
- Accuracy
- Speed
- Meta-data overhead

Least Recently Used (LRU)

- Maintain a timestamp of when each page was last accessed.
- DBMS picks to evict the page with the oldest timestamp.

CLOCK

Approximation of LRU without needing a separate timestamp per page.

- Each page has a reference bit
- When a page is accessed, set to 1

Organize the pages in a circular buffer with a "clock hand"

- Upon sweeping check if a pages bit is set to 1
- If yes, set to zero, if no, then evict
- Clock hand remembers position between evictions

Alternatives

Problems with LRU and Clock replacement policies:

- LRU and Clock are susceptible to **sequential flooding** where the buffer pool's contents are trashed due to a sequential scan.
- It may be that the LRU page is actually important due to not tracking meta-data of how a page is used.

Better solutions:

- LRU-K: Take into account history of the last K references
- Priority hints: Allow txns to tell the buffer pool whether page is important or not
- Localization: Choose pages to evict on a per txn/query basis

Lecture #06: Hash Tables

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Data Structures

A DBMS uses various data structures for many different parts of the system internals:

- **Internal Meta-Data:** Keep track of information about the database and the system state.
- **Core Data Storage:** Can be 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 Indexes:** Auxiliary data structures to make it easier to find specific tuples.

Design Decisions:

1. Data organization: How we layout memory and what information to store inside the data structure.
2. Concurrency: 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 and $O(n)$ storage complexity.

A hash table implementation is comprised of two parts:

- **Hash Function:** How to map a large key space into a smaller domain. This is used to compute an index into an array of buckets or slots. Need to consider the trade-off between fast execution vs. collision rate.
- **Hashing Scheme:** How to handle key collisions after hashing. Need to consider the trade-off between the need to allocate a large hash table to reduce collisions vs. executing additional instructions to find/insert keys.

3 Hash Functions

A *hash function* takes in any key as its input. It then return 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 does not want to use a cryptography 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. For this lecture, 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 it from scratch with a larger table. 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. This requires hash table with twice the number of slots as the number of expected elements.

4.1 Linear Probe Hashing

This is the most basic hashing scheme. It is also typically the fastest. It uses a single table of slots. The hash function allows the DBMS to quickly jump to slots and look for the desired key.

- Resolve collisions by linearly searching for the next free slot in the table
- To see if value is present, go to slot using hash, and scan for the key. The scan stops if you find the desired key or you encounter an empty slot.

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 in the hash table. Allows threads to steal slots from "rich" keys and give them to "poor" keys.

- Each key tracks the number of positions they are from where its optimal position in the table.
- On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key. The removed key then has to be re-inserted back into 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 function are the same algorithm (e.g., XXHash, CityHash); they generate different hashes for the same key by using different seed values.

- On insert, check every table and pick anyone that has a free slot.
- If no table has free slot, evict element from one of them, and rehash it to find a new location.
- If we find a cycle, then 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).

5 Dynamic Hashing Schemes

The static hashing schemes require the DBMS to know the number of elements it wants to store. Otherwise it has 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.

- Resolves collisions by placing elements with same hash key into the same bucket.

- If bucket is full, add another bucket to that chain. The hash table can grow infinitely because the DBMS keeps adding new buckets.

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 the 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 hash function maps to slot that has previously been pointed to by pointer, apply the new hash function.
- When pointer reaches last slot, delete original hash function and replace it with new hash function.

Lecture #07: Tree Indexes I

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Indexes

This lecture continues our discussion of data structures. We are going to focus on table indexes.

A *table index* is a replica of a subset of a table's columns that is organized in such a way that allows the DBMS to find tuples more quickly than performing a sequential scan. The DBMS ensures that the contents of the tables and the indexes are always in sync.

It is the DBMS's job to figure out the best indexes to use to execute queries. There is a trade-off on the number of indexes to create per database (indexes use storage and require maintenance).

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 DBMSs 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. Modern B+Tree implementations combine features from other B-Tree variants, such as the sibling pointers used in the B^{link}-Tree.

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:

- Arrays at every node are (almost) sorted by the keys.
- The value array for inner nodes will contain pointers to other nodes.
- Two approaches for leaf node values
  1. Record IDs: A pointer to the location of the tuple
  2. Tuple Data: The actual contents of the tuple is stored in the leaf node

2.1 Insertion

1. Find correct leaf L.
2. Add new entry into L in sorted order:
   - If L has enough space, the operation 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.

2.2 Deletion

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.

3 B+Tree Design Decisions

Node Size:

- The optimal node size for a B+Tree depends on the speed of the disk. The idea is to amortize the cost of reading a node from disk into memory over as many key/value pairs as possible.
- The slower the disk, then the larger the idea node size.
- Some workloads may be more scan-heavy versus having more single key look-ups.

Merge Threshold:

- Some DBMS do not always merge when it's half full.
- Delaying a merge operation may reduce the amount of reorganization.
- It may be better for the DBMS to let underflows to occur and then periodically rebuild the entire tree to re-balance it.

Variable Length keys:

- Pointers: Store keys as pointers to the tuples attribute (very rarely used).
- Variable-length Nodes: The size of each node in the B+Tree can vary, but requires careful memory management. This approach is also rare.
- Key Map: Embed an array of pointers that map to the key+value list within the node. This is similar to slotted pages discussed before. This is the most common approach.

Non-Unique Indexes:

- Duplicate Keys: Use the same leaf node layout but store duplicate keys multiple times.
- Value Lists: Store each key only once and maintain a linked list of unique values.

Intra-Node Search:

- Linear: Scan the key/value entries in the node from beginning to end. Stop when you find the key that you are looking for. This does not require the key/value entries to be pre-sorted.
- Binary: Jump to the middle key, and then pivot left/right depending on whether that middle key is less than or greater than the search key. This requires the key/value entries to be pre-sorted.
- Interpolation: Approximate the starting location of the search key based on the known low/high key values in the node. Then perform linear scan from that location. This requires the key/value entries to be pre-sorted.

4 B+Tree Optimizations

Prefix Compression:

- Sorted keys in the same leaf node are likely to have the same prefix.
- Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.

Suffix Truncation:

- The keys in the inner nodes are only used to “direct traffic”, we do not need the entire key.
- Store a minimum prefix that is needed to correctly route probes into the index.

Bulk Inserts:

- The fastest way to build a B+Tree from scratch is to first sort the keys and then build the index from the bottom up.
- This will be faster than inserting one-by-one since there are no splits or merges.

Lecture #08: Tree Indexes II

15-445/645 Database Systems (Fall 2019)

`https://15445.courses.cs.cmu.edu/fall2019/`

Carnegie Mellon University
Prof. Andy Pavlo

1 Additional Index Usage

Implicit Indexes: Most DBMSs will automatically create an index to enforce integrity constraints (e.g., primary keys, unique constraints).

Partial Indexes: Create an index on a subset of the entire table. This potentially reduces size and the amount of overhead to maintain it.

Covering Indexes: All attributes needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple. The DBMS can complete the entire query just based on the data available in the index.

Index Include Columns: Embed additional columns in index to support index-only queries.

Function/Expression Indexes: Store the output of a function or expression as the key instead of the original value. It is the DBMS's job to recognize which queries can use that index.

2 Radix Tree

A *radix tree* is a variant of a trie data structure. It uses digital representation of keys to examine prefixes one-by-one instead of comparing entire key. It is different than a trie in that there is not a node for each element in key, nodes are consolidated to represent the largest prefix before keys differ.

The height of tree depends on the length of keys and not the number of keys like in a B+Tree. The path to a leaf nodes represents the key of the leaf. Not all attribute types can be decomposed into binary comparable digits for a radix tree.

3 Inverted Indexes

An *inverted index* stores a mapping of words to records that contain those words in the target attribute. These are sometimes called a *full-text search indexes* in DBMSs.

Most of the major DBMSs support inverted indexes natively, but there are specialized DBMSs where this is the only table index data structure available.

Query Types:

- Phrase Searches: Find records that contain a list of words in the given order.
- Proximity Searches: Find records where two words occur within n words of each other.
- Wildcard Searches: Find records that contain words that match some pattern (e.g., regular expression).

Design Decisions:

- What To Store: The index needs to store at least the words contained in each record (separated by punctuation characters). It can also include additional information such as the word frequency, position, and other meta-data.

- When To Update Updating an inverted index every time the table is modified is expensive and slow. Thus, most DBMSs will maintain auxiliary data structures to "stage" updates and then update the index in batches.

Lecture #09: Index Concurrency Control

15-445/645 Database Systems (Fall 2019)

`https://15445.courses.cs.cmu.edu/fall2019/`

Carnegie Mellon University

Prof. Andy Pavlo

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:** Can I see the data that I am supposed to see? This means that the thread is able to read values that it should be allowed to read.
- **Physical Correctness:** Is the internal representation of the object sound? This means that there are not pointers in our 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

Locks:

- Protects the indexs logical contents from other transactions.
- Held for (mostly) the entire duration of the transaction.
- The DBMS needs to be able to rollback changes.

Latches:

- Protects the critical sections of the indexs internal data structure from other threads.
- Held for operation duration.
- The DBMS does not need to be able to rollback changes.
- Two Modes:
  - **`READ`:** Multiple threads are allowed to read the same item at the same time. A thread can acquire the read latch if another thread has it in read mode.
  - **`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.

3 Latch Implementations

The underlying primitive that we can use 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 have different tradeoffs 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 after one thread checks it but before it updates it).

Blocking OS Mutex

Use the OS built-in mutex infrastructure as a latch. The futex (fast user-space mutex) is comprised of (1) a spin latch in user-space and (2) a 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 lock 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. If it cannot, then it spins in a while loop forever trying to update it.

- **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). We need 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.

- **Example:** This is implemented on top of Spin Latches.
- **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. To resize the table, take a global latch on the entire table (i.e., in the header page).

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

In general, there are two approaches to support latching in a hash table:

- **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 because a thread 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 in 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. The DBMS can use a single mode latch (i.e., Spin Latch) to reduce meta-data and computational overhead.

5 B+Tree Latching

Lock crabbing / coupling is a protocol to allow multiple threads to access/modify B+Tree at the same time:

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

Basic Latch Crabbing Protocol:

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

Improved Lock 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, we 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 any node in the path is not safe, then do previous algorithm (i.e., 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 leaf is not safe, release all previous latches, and restart transaction using previous Insert/Delete protocol.

6 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 we can deal with this problem is through coding discipline. The leaf node sibling latch acquisition protocol must support a "no-wait" mode. That is, 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 #10: Sorting & Aggregation Algorithms

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Sorting

We need sorting because in the relation model, tuples in a table have no specific order Sorting is (potentially) used in ORDER BY, GROUP BY, JOIN, and DISTINCT operators.

We can accelerate sorting using a *clustered* B+tree by scanning the leaf nodes from left to right. This is a bad idea, however, if we use an *unclustered* B+tree to sort because it causes a lot of I/O reads (random access through pointer chasing).

If the data that we need to sort 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.

2 External Merge Sort

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.

Phase #1 – Sorting: Sort small chunks of data that fit in main memory, and then write back to disk.

Phase #2 – Merge: Combine sorted sub-files into a larger single file.

Two-way Merge Sort

1. Pass #0: Reads every B pages of the table into memory. Sorts them, and writes them back into disk. Each sorted set of pages is called a **run**.
2. Pass #1,2,3...: Recursively merges pairs of runs into runs twice as long.

Number of Passes: $1 + \lceil log_2 N \rceil$
Total I/O Cost: $2N \times$ (# of passes)

General (K-way) Merge Sort

1. Pass #0: Use B buffer pages, produce N/B sorted runs of size B.
2. Pass #1,2,3...: Recursively merge $B-1$ runs.

Number of Passes = $1 + \left\lceil log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil$
Total I/O Cost: $2N \times$ (# of passes)

Double Buffering Optimization

Prefetch the next run in the background and store 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.

3 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.

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:

- **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. Note that 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 #11: Joins Algorithms

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Joins

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

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 processing model, storage model, and the query itself:

- **Data:** Copy 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.
- **Record Ids:** 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 that we are going to use 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 intermediate data out to disk.

Variables used in this lecture:
- M pages in table R, m tuples total
- N pages in table S, n tuples total

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. If possible, 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 you assume that there is one disk I/O to read each tuple (i.e., there is no 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 we scan 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 hold a block from 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. But if the database already has an index for one of the tables on the join key, then it can use that to speed up the comparison. 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

The high-level is to sort the two tables on their join key. Then perform a sequential scan on the sorted tables to compute the join. This algorithm is useful if one or both tables are already sorted on join attribute(s).

The worst case scenario for this algorithm is if the join attribute for all the tuples in both tables contain the same value. This is very unlikely to happen in real databases.

- **Phase #1 – Sort:** First sort both input tables on the join attribute.
- **Phase #2 – Merge:** Scan the two sorted tables in parallel, and emit matching tuples.

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

- Sort Cost for Table R: $2M \times 1 + \left\lceil log_{B-1} \left\lceil \frac{M}{B} \right\rceil \right\rceil$
- Sort Cost for Table S: $2N \times 1 + \left\lceil log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil$
- 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 in bucket

s_i. Thus, R tuples in bucket r_i need only to be compared with S tuples in bucket s_i.

Basic Hash Join

- **Phase #1 – Build:** 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.
- **Phase #2 – Probe:** Scan the inner relation and use the hash function h_1 on each tuple to jump to a 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.

Grace Hash Join / Hybrid Hash Join

When the tables do not fit on main memory, you do not want the buffer pool manager constantly swapping tables in and out. The Grace Hash Join is an extension of the basic hash join that is also hashes the inner table into partitions that are written out to disk. The name “Grace” comes from GRACE database machine developed during the 1980s in Japan.

- **Phase #1 – Build:** 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, then use *recursive partitioning* with a second hash function h_2 (where $h_1 \neq h_2$) to further divide the bucket.
- **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: $3 \times (M + N)$

Lecture #12: Query Processing I

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Query Plan

The DBMS converts a SQL statement into a query plan. Operators are arranged in a tree. Data flows from the leaves 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. Most DBMSs will want to use an index scan as much as possible.

2 Processing Models

A DBMS *processing model* defines how the system executes a query plan. There are different models that have various trade-offs for different workloads.

These models can also be implemented to invoke the operators either from **top-to-bottom** (most common) or from **bottom-to-top**.

Iterator Model

This is the most common processing model and is used by almost every (row-based) DBMS. Allows for *pipelining* where the DBMS can process a tuple through as many operators as possible before having to retrieve the next tuple.

Every query plan operator implements a `next` function:

- On each call to `next`, the operator returns either a single tuple or a null marker if there are no more tuples.
- The operator implements a loop that calls `next` on its children to retrieve their tuples and then process them (i.e., calling `next` on a parent calls `next` on their children).

Some operators will block until children emit all of their tuples (joins, subqueries, order by). These are known as *pipeline breakers*.

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

Materialization Model

Each operator processes its input all at once and then emits its output all at once. The operator “materializes” its output as a single result.

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. Not good 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 where each operator implements a `next` function. But each operator emits a *batch* (i.e., vector) of data instead of a single tuple:

- The operator implementation can be optimized for processing batches of data instead of a single item at a time.

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

3 Access Methods

An access method is the how the DBMS accesses the data stored in a table. These will be the bottom operators in a query plan that "feed" data into the operators above it in the tree. There is no corresponding operator in relational algebra.

Sequential Scan

For each page in table, iterate over each page and retrieve it from the buffer pool. For each page, iterate over all the tuples and evaluate the predicate to decide whether to include tuple or not.

Optimizations:

- **Prefetching:** Fetches next few pages in advance so that the DBMS does not have to block when accessing each page.
- **Parallelization:** Execute the scan using multiple threads/processes in parallel.
- **Buffer Pool Bypass:** The scan operator stores pages that it fetches from disk in its local memory instead of the buffer pool. This avoids the sequential flooding problem.
- **Zone Map:** Pre-compute aggregations for each tuple attribute in a page. The DBMS can then check 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.
- **Late Materialization:** Each operator passes the minimal amount of information needed to by the next operator (e.g., record id). This is only useful in column-store systems (i.e., DSM).
- **Heap Clustering:** Tuples are stored in the heap pages using an order specified by a clustering index.

Index Scan

The DBMS picks an index (or indexes) to find the tuples that the query needs.

When using multiple indexes, the DBMS executes the search on each index and generates the set of matching record ids. One can implement this record id using bitmaps, hash tables, or Bloom filters. The DBMS combines these sets based on the query's predicates (union vs. intersect). It then retrieve the records and apply any remaining terms. The more advanced DBMSs support multi-index scans.

Retrieving tuples in the order that they appear in 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.

4 Expression Evaluation

The DBMS represents a query plan as a tree. Inside of the operators will be an expression tree. For example, the `WHERE` clause for a filter operator.

The nodes in the tree represent different expression types:

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

Lecture #13: Query Execution II

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Background

Our previous discussion of query execution did not specify how the DBMS would compute results. We are now going to discuss how to organize the system's internals on the CPU.

The main goal is to understand how to enable the DBMS to support *parallel* query execution. This provides several benefits:

- Increased performance in throughput and latency.
- Increased availability.
- 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.

The techniques discussed here are applicable to distributed DBMSs as well, so we first need to understand the differences between a *parallel* and *distributed* DBMS. These systems spread the database out across multiple "resources" to improve parallelism. These resources are either computational (e.g., CPU cores, CPU sockets, GPUs, additional machines) or storage (e.g., disks, memory).

Parallel DBMS:

- Nodes are physically close to each other.
- Nodes are connected with high-speed LAN.
- Communication cost between nodes is assumed to be fast and reliable.

Distributed DBMS:

- Nodes can be far from each other.
- Nodes are connected using public network.
- Communication costs between nodes is slower and failures cannot be ignored.

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

Types of Parallelism

- **Inter-Query:** The DBMS executes different queries are concurrently. This increases throughput and reduces latency. Concurrency is tricky when queries are updating the database.
- **Intra-Query:** The DBMS executes the operations of a single query in parallel. This decreases latency for long-running queries.

2 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.

Approach #1 – Process per Worker:

- Each worker is a separate OS process, and thus relies on OS scheduler.
- Use shared memory for global data structures.
- A process crash does not take down entire system.

Approach #2 – Process Pool:

- A worker uses any process that is free in a pool.
- Still relies on OS scheduler and shared memory.
- This approach can be bad for CPU cache locality due to no guarantee of using the same process between queries.

Approach #3 – Thread per Worker:

- Single process with multiple worker threads.
- DBMS has to manage its own scheduling.
- May or may not use a dispatcher thread.
- Although a thread crash (may) kill the entire system, we have to make sure that we write high-quality code to ensure that this does not happen.

Using a multi-threaded architecture has advantages that there is less overhead per context switch and you do not have to manage shared model. The thread per worker model does not mean that you have intra-query parallelism.

For each query plan, the DBMS has to decide where, when, and how to execute:

- 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?

3 Inter-Query Parallelism

The goal of this type of parallelism is to improve the DBMS's overall performance by allowing multiple queries to execute simultaneously.

We will cover this in more detail when we discuss concurrency control protocols.

4 Intra-Query Parallelism

The goal of this type of parallelism is to improve the performance of a single query by executing its operators in parallel. There are parallel algorithms for every relational operator.

Intra-Operator Parallelism

The query plan's operators are decomposed into independent instances that perform the same function on different subsets of data.

The DBMS inserts an **exchange** operator into the query plan to coalesce results from children 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.

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

The DBMS overlaps operators in order to pipeline data from one stage to the next without materialization. This is sometimes called **pipelined parallelism**.

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

Bushy Parallelism

Extension of inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.

The DBMS still uses exchange operators to combine intermediate results from these segments.

5 I/O Parallelism

Using additional processes/threads to execute queries in parallel will not improve performance if the disk is always the main bottleneck. Thus, we need a way to split the database up across multiple storage devices.

Multi-Disk Parallelism

Configure OS/hardware to store the DBMS's files across multiple storage devices. Can be done through storage appliances and RAID configuration. This is transparent to the DBMS. It cannot have workers operate on different devices because it is unaware of the underlying parallelism.

File-based Partitioning

Some DBMSs allow you to specify the disk location of each individual database. The buffer pool manager maps a page to a disk location. This is also easy to do at the file-system level if the DBMS stores each database in a separate directory. However, the log file might be shared.

Logical Partitioning

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.

Vertical Partitioning:

- Store a table's attributes in a separate location (like a column store).
- Have to store tuple information to reconstruct the original record.

Horizontal Partitioning:

- Divide the tuples of a table 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 #14: Query Planning & Optimization I

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Overview

SQL is declarative. This means that the user tells the DBMS what answer they want, not how to get the answer. Thus, the DBMS needs to translate a SQL statement into an executable query plan. But there are different ways to execute a query (e.g., join algorithms) and there will be differences in performance for these plans. Thus, the DBMS needs a way to pick the "best" plan for a given query. This is the job of the DBMS's optimizer.

There are two types of optimization strategies:

- **Heuristics/Rules:** Rewrite the query to remove inefficiencies. Does not require a cost model.
- **Cost-based Search:** Use a cost model to evaluate multiple equivalent plans and pick the one with the smallest cost.

2 Rule-based Query Optimization

Two relational algebra expressions are equivalent if they generate the same set of tuples. Given this, the DBMS can identify better query plans without a cost model. This is technique often called **query rewriting**. Note that most DBMSs will rewrite the query plan and not the raw SQL string.

Examples of query rewriting:

- **Predicate Push-down:** Perform predicate filtering before join to reduce size of join).
- **Projections Push down:** Perform projections early to create smaller tuples and reduce intermediate results. You can project out all attributes except the ones requested or required (e.g. join attributes).
- **Expression Simplification:** Exploit the transitive properties of boolean logic to rewrite predicate expressions into a more simple form.

Lecture #15: Query Planning & Optimization II

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Cost-based Query Optimization

The DBMS's optimizer will use an internal *cost model* to estimate the execution cost for a particular query plan. This provides an estimate to determine whether one plan is better than another without having to actually run the query (which would be slow to do for thousands of plans).

This estimate is an internal metric that (usually) is not comparable to real-world metrics, but it can be derived from estimating the usage of different resources:

- **CPU:** Small cost; tough to estimate.
- **Disk:** Number of block transferred.
- **Memory:** Amount of DRAM used.
- **Network:** Number of messages transfer ed.

To accomplish this, the DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog. Different systems update the statistics at different times. Commercial DBMS have way more robust and accurate statistics compared to the open source systems. These are estimates and thus the cost estimates will often be inaccurate.

2 Statistics

For a relation R, the DBMS stores the number of tuples (N_R) and distinct values per attribute ($V(A, R)$).

The *selection cardinality* ($SC(A, R)$) is the average number of records with a value for an attribute A given $N_R/V(A, R)$.

Complex Predicates

- The *selectivity* (sel) of a predicate P is the fraction of tuples that qualify:

$$sel(A = constant) = SC(P)/V(A, R)$$

- For a range query, we can use: $sel(A >= a) = (A_{max} - a/(A_{max} - A_{min}))$.
- For negations: $sel(notP) = 1 - sel(P)$.
- The selectivity is the probability that a tuple will satisfy the predicate. Thus, assuming predicates are independent, then $sel(P1 \bigwedge P2) = sel(P1) * sel(P2)$.

Join Estimation

- Given a join of R and S, the estimated size of a join on non-key attribute A is approx

$$estSize \approx N_R * N_S/max(V(A, R), V(A, S))$$

Statistics Storage

Histograms: We assumed values were uniformly distributed. But in real databases values are not uniformly distributed, and thus maintaining a histogram is expensive. We can put values into buckets to reduce the size of the histograms. However, this can lead to inaccuracies as frequent values will sway the count of infrequent values. To counteract this, we can size the buckets such that their spread is the same. They each hold a similar amount of values.

Sampling: Modern DBMSs also employ **sampling** to estimate predicate selectivities. Randomly select and maintain a subset of tuples from a table and estimate the selectivity of the predicate by applying the predicate to the small sample.

3 Search Algorithm

The basic cost-based search algorithm for a query optimizer is the following:

1. Bring query in internal form into canonical form.
2. Generate alternative plans.
3. Generate costs for each plan.
4. Select plan with smallest cost.

It is important to pick the best access method (i.e., sequential scan, binary search, index scan) for each table accessed in the query. Simple heuristics are sometimes good enough for simple OLTP queries (i.e., queries that only access a single table). For example, queries where it easy to pick the right index to use are called *sargable* (Search Argument Able). Joins in OLTP queries are also almost always on foreign key relationships with small cardinality.

For multiple relation query planning, the number of alternative plans grows rapidly as number of tables joined increases. For an n-way join, the number of different ways to order the join operations is known as a Catalan number (approx 4^n). This is too large of a solution space and it is infeasible for the DBMS to consider all possible plans. Thus, we need a way to reduce the search complexity. For example, in IBM's System R, they only considered *left-deep* join trees. Left-deep joins allow you to pipeline data, and only need to maintain a single join table in memory.

4 Nested Sub-Queries

The DBMS treats nested sub-queries in the `WHERE` clause as functions that take parameters and return a single value or set of values.

Two Approaches:

1. Rewrite to decorrelate and/or flatten queries.
2. Decompose nested query and store result in sub-table.

Lecture #16: Concurrency Control Theory

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

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.

Example: Move $100 from Andy's bank account to his bookie's account

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

Executing concurrent transactions in a DBMS is challenging. It is difficult to ensure correctness while also executing transactions quickly. We need formal correctness criteria:

- Temporary inconsistency is allowed.
- Permanent inconsistency is bad.

The scope of a transaction is only inside the database. It cannot make changes to the outside world because it cannot roll those back.

2 Definitions

A database is a set of named data objects $(A, B, C, ..)$. A transaction is a sequence of read and write operations $(R(A), W(B))$.

The outcome of a transaction is either `COMMIT` or `ABORT`.

- If `COMMIT`, all of the transaction's modifications are saved to the database.
- If `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.

Correctness Criteria: **ACID**

- **A**tomicity: All actions in the transaction happen, or none happen.
  "All or Nothing"
- **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.
  "It looks correct to me..."
- **I**solation: The execution of one transaction is isolated from that of other transactions.
  "As if alone"
- **D**urability: If a transaction commits, then its effects on the database persist.
  "The transaction's changes can survive failures..."

3 ACID: Atomicity

The DBMS guarantees that transactions are **atomic**. The transaction either executes all its actions or none of them.

Approach #1: Shadow Paging

- DBMS makes copies of pages and transactions make changes to those copies. Only when the transaction commits is the page made visible to others.
- Originally from System R but abandoned in the early 1980s. Few systems do this today (CouchDB, LMDB).

Approach #1: Logging

- DBMS logs all actions so that it can undo the actions of aborted transactions.
- Think of this like the black box in airplanes.
- Logging is used by all modern systems for audit and efficiency reasons.

4 ACID: Consistency

The “world” represented by the database is **consistent** (e.g., correct). All questions (i.e., queries) that the application asks about the data will return correct results.

Database Consistency:

- The database accurately represents the real world entity it is modeling and follows integrity constraints.
- Transactions in the future 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 in order to get better performance, the DBMS has to interleave the operations of concurrent transactions.

Concurrency Control

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

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.

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:** A schedule that does not interleave the actions of different transactions.
- **Equivalent Schedules:** For any database state, the effect of execution the first schedule is identical to the effect of executing the second schedule.
- **Serializable Schedule**: A schedule that is equivalent to some serial execution of the transactions.

When the DBMS interleaves the operations of concurrent transactions, it can create anomalies:

- **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 actually two types for serializability: (1) conflict and (2) view. Neither definition allows all schedules that you would consider serializable. In practice, DBMSs support conflict serializability because it can be enforced efficiently. To allow more concurrency, some special schedules are handled at the application level.

Conflict Serializability

Schedules are equivalent to some serial schedule. This is what (almost) every DBMS supports when you ask for the `SERIALIZABLE` isolation level.

Schedule S is conflict serializable if you are able to transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions.

Verify using either the swapping method or dependency graphs.

Dependency Graphs (aka "precedence graph"):

- One node per transaction.
- Edge from T_i to T_j if an operation O_i of T_i conflicts with an operation O_j of T_j and O_i appears earlier in the schedule than O_j.
- A schedule is conflict serializable if and only if its dependency graph is acyclic.

View Serializability

Allows for all schedules that are conflict serializable and "blind writes". Thus allows for slightly more schedules than Conflict serializability, but difficult to enforce efficiently. This is because the DBMS does not now 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 #17: Two-Phase Locking

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Transaction Locks

The DBMS contains a centralized lock manager that decides decisions whether a transaction can have a lock or not. It has a global view of whats going on inside the system.

- **Shared Lock (`S-LOCK`):** A 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`):** Allows a transaction to modify an object. This lock is not compatible for any other lock. Only one transaction can hold an exclusive lock at a time.

Executing with locks:

1. Transactions request locks (or upgrades) from the lock manager.
2. The lock manager grants or blocks requests based on what locks are currently held by other transactions.
3. Transactions release locks when they no longer need them.
4. The lock manager updates its internal lock-table and then gives locks to waiting transactions.

2 Two-Phase Locking

Two-Phase locking (2PL) is a pessimistic concurrency control protocol that determines 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

- Each transaction requests the locks that it needs from the DBMS's lock manager.
- The lock manager grants/denies lock requests.

Phase #2: Shrinking

- The transaction enters this phase immediately after it releases its first lock.
- The transaction is allowed to only release locks that it previously acquired. It cannot acquire new locks in this phase.

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).

3 Strong Strict Two-Phase Locking

Strong Strict 2PL (SSPL, also known as Rigorous 2PL) is a variant of 2PL where the transaction only releases locks when it finishes. A schedule is *strict* if a value written by a transaction is not read or overwritten by other transactions until that transaction finishes. Thus, there is not a shrinking phase in SS2PL like in regular 2PL.

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 just restoring original values of modified tuples.

4 2PL 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

The DBMS creates a *waits-for* graph: Nodes are transactions, and 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 waits-for graph and then make a decision on how to break it.

- When the DBMS detects a deadlock, it will select a "victim" transaction to rollback to break the cycle.
- The victim transaction will either restart or abort depending on how the application invoked it
- There are multiple transaction properties to consider when selecting a victim. There is no one choice that is better than others. 2PL DBMSs all do different things:
  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 that we have to rollback with it.
  5. # of times a transaction has been restarted in the past
- **Rollback Length:** After selecting a victim transaction to abort, the DBMS can also decide on how far to rollback the transaction's changes. Can be either the entire transaction or just enough queries to break the deadlock.

Approach #2: Deadlock Prevention

When a transaction tries to acquire a lock, if that lock is currently held by another transaction, then perform some action to prevent a deadlock. Assign priorities based on timestamps (e.g., older means higher priority). These schemes guarantee no deadlocks because only one type of direction is allowed when waiting for a lock. When a transaction restarts, its (new) priority is its old timestamp.

- **Wait-Die ("Old waits for Young"):** If T_1 has higher priority, T_1 waits for T_2. Otherwise T_1 aborts
- **Wound-Wait ("Young waits for Old"):** If T_1 has higher priority, T_2 aborts. Otherwise T_1 waits.

5 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 a transaction acquires a lock for an object in this hierarchy, it implicitly acquires the locks for all its children.

Intention locks allow a higher level node to be locked in shared 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.

6 Conclusion

2PL is used in most DBMSs that support transactions. The protocol automatically provides correct interleavings of transaction operations, but it requires additional steps to handle deadlocks.

The application does not typically set locks manually using SQL. The DBMS acquires the locks automatically before a query accesses or modifies an object. But sometimes the application can provide the DBMS with hints to help it improve concurrency:

`SELECT...FOR UPDATE`: Perform a select and then sets an exclusive lock on fetched tuples

Lecture #18: Timestamp Ordering

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Timestamp Ordering Concurrency Control

Timestamp ordering (T/O) is a 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 that is monotonically increasing:

- Let TS(T_i) be the timestamp allocated to transaction T_i
- Different schemes assign timestamps at different times during the 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.

Multiple timestamp allocation implementation strategies:

- System clock
- Logical counter
- Hybrid

2 Basic Timestamp Ordering (BASIC T/O)

Every database object X is tagged with timestamp of the last transaction that successfully did read/write:

- W-TS(X): Write timestamp on object X.
- R-TS(X): Read timestamp on object X.

The DBMS check timestamps for every operation. If transaction tries to access an object "from the future", then the DBMS aborts that transaction and restarts it.

Read Operations:

- If TS(T_i) $<$ W-TS(X) this violates timestamp order of T_i with regard to the writer of X. Thus you abort T_i and restart it with same TS.
- Else:
  - Allow T_i to read X.
  - Update R-TS(X) to max(R-TS(X), TS(T_i)).
  - Have to make a local copy of X to ensure repeatable reads for T_i.
  - Last step may be skipped in lower isolation levels.

Write Operations:

- If TS(T_i) $<$ R-TS(X) or TS(T_i) $<$ W-TS(X), abort and restart T_i.
- Else:

  - Allow T_i to write X and update W-TS(X) to T_i.
  - Also have to make a local copy of X to ensure repeatable reads for T_i.

Optimization: Thomas Write Rule

- If TS(T_i) $<$ R-TS(X): Abort and restart T_i
- If TS(T_i) $<$ W-TS(X):
  - **Thomas Write Rule:** Ignore the write and allow transaction to continue.
  - 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.
- Else: Allow T_i to write X and update W-TS(X)

The Basic T/O protocol generates a schedule that is conflict serializable if you do not use Thomas Write Rule. It cannot have deadlocks because no transaction ever waits. But 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 or 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)

If we assume that conflicts between transactions are rare and most transactions are short lived, it may be a better approach to optimize for the common case that assumes transactions are not going to have conflicts.

OCC works well 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, so again locking is wasteful

The DBMS creates a **private workspace** for each transaction:

- All modifications are applied to the workspace.
- Any object read is copied into workspace.
- 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 Transaction Phases:

- **Read Phase:** Track the read/write sets of transactions and store their writes in a private workspace.
- **Validation Phase:** When a transaction commits, check whether it conflicts with other transactions.
- **Write Phase:** If validation succeeds, apply private changes to database. Otherwise abort and restart the transaction.

Validation Phase

This is where the DBMS checks whether a transaction conflicts with other transactions. The DBMS needs to guarantee that only serializable schedules are permitted. The DBMS assigns transactions timestamps when they enter the validation phase.

T_i checks 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:

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 Partition-Based T/O

When a transaction commits in OCC, the DBMS has check whether there is a conflict with concurrent transactions across the entire database. This is slow if we have a lot of concurrent transactions because the DBMS has to acquire latches to do all of these checks.

An alternative is to split the database up in disjoint subsets called **partitions** (aka shards) and then only check for conflicts between transactions that are running in the same partition.

Partitions are protected by a single lock. Transactions are assigned timestamps based on when they arrive at the DBMS. Each transaction is queued at the partitions it needs before it starts running:

- The transaction acquires a partition's lock if it has the lowest timestamp in that partition's queue.
- The transaction starts when it has all of the locks for all the partitions that it will access during execution.
- Transactions can read/write anything that they want at the partitions that they have locked. If a transaction tries to access a partition that it does not have the lock, it is aborted + restarted.

Potential Issues:

- Partition-based T/O protocol is fast if: (1) the DBMS knows what partitions the transaction needs before it starts and (2) most (if not all) transactions only need to access a single partition.
- The protocol only works if (1) transactions are stored procedures (network communication causes the partition to idle because it has to wait for the next query to execute) and (2) transactions only touch one partition (multi-partition transactions cause partitions to be idle because partitions have to wait for the next query to execute).

Lecture #19: Multi-Version Concurrency Control

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) is a larger concept than just a concurrency control protocol. It involves all aspect of the DBMS's design and implementation. MVCC is the most widely used scheme in DBMS. 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.

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 original MVCC protocol for DBMSs was first proposed in 1978 MIT PhD dissertation. The first implementation in a real DBMS was in InterBase (now open-sourced as Firebird) by Jim Starkey, who later went on to be the co-founder of NuoDB.

Key Properties

Writers don't block the readers. Readers don't block the writers.

Read-only transactions can read a consistent **snapshot** without acquiring locks. Timestamps are used to determine visibility.

Multi-versioned DBMSs can support *time-travel queries* that can read the database at a point-in-time snapshot.

There are four important MVCC design decisions:

1. Concurrency Control Protocol (T/O, OCC, 2PL, etc).
2. Version Storage
3. Garbage collection
4. Index Management

2 Version Storage

This how the DBMS will store the different physical versions of a logical object.

The DBMS uses the tuple's pointer field to create a **version chain** per logical tuple. 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. A thread traverses chain until you find the version that is visible to you. Different storage schemes determine where/what to store for each version.

Approach #1: Append-Only Storage – New versions are appended to the same table space.

- **Oldest-To-Newest (O2N):** Append new version to end of chain, look-ups require entire chain traversal.
- **Newest-To-Oldest (N2O):** Head of chain is newest, look-ups are quick, but indexes need to be updated every version.

Approach #2: Time-Travel Storage – Old versions are copied to separate table space.

Approach #3: Delta Storage – The original values of the modified attributes are copied into a separate delta record space.

3 Garbage Collection

The DBMS needs to remove **reclaimable** physical versions from the database over time.

Approach #1: Tuple Level Garbage Collection – Find old versions by examining tuples directly

- **Background Vacuuming**: Separate threads periodically scan the table and look for reclaimable versions, works with any version storage scheme.
- **Cooperative Cleaning**: Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

Approach #2: Transaction Level – Each transaction keeps track of its own read/write set. When a transaction completes, the garbage collector can use that to identify what 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:

- **Approach #1: Logical Pointers** – Use a fixed identifier per tuple that does not change. Requires an extra indirection layer that maps the logical id to the physical location of the tuple (Primary Key vs Tuple ID).
- **Approach #2: Physical Pointers** – Use the physical address to the version chain head

Lecture #20: Logging Schemes

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Crash Recovery

Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures. DBMS is divided into different components based on the underlying storage device. We must also classify the different types of failures that the DBMS needs to handle.

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 to recover the database to a state that ensures atomicity, consistency, and durability.

The key primitives that we are going to use in a recovery algorithm are UNDO and REDO. Not all algorithms use both of these:

- **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 Failure Classification

Type #1: Transaction Failures

- **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

- **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. We assume that non-volatile storage contents are not corrupted by system crash.

Type #3: Storage Media Failure

- **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. No DBMS can recover from this. Database must be restored from archived version

3 Buffer Pool Management Policies

Steal Policy: 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.

Force Policy: Whether the DBMS ensures that all updates made by a transaction are reflected on non-volatile storage before the transaction is allowed to commit

- **FORCE**: Is enforced
- **NO-FORCE**: Is not enforced

Force writes makes it easier to recover but results in poor runtime performance.

Easiest System to implement: NO-STEAL + FORCE

- 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 committed.
- **Limitation:** If all of the data that a transaction needs to modify does not fit on memory, then that transaction cannot execute because the DBMS is not allowed to write out dirty pages to disk before the transaction commits.

4 Shadow Paging

The DBMS maintains two separate copies of the database (**master, shadow**). Updates are only made in the shadow copy. When a transaction commits, atomically switch the shadow to become the new master. This is an example of a NO-STEAL + FORCE system.

Implementation:

- Organize 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 the shadow:
  - The root points to the master copy
  - Updates are applied to the shadow copy
- To install updates, overwrite the root so it points to the shadow, 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.
- **UNDO:** Remove the shadow pages. Leave master and the DB root pointer alone
- **REDO:** Not needed at all

5 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. This is an example of a STEAL + NO-FORCE system.

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.

Implementation:

- All log records pertaining to an updated page are 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).
- Log entries to disk should be done when transaction commits. You can use group commit to batch multiple log flushes together to amortize overhead.

6 Checkpoints

The main problem with write-ahead logging is that the log file will grow forever. After a crash, the DBMS has to replay the entire log, which can take a long time if the log file is large. Thus, the DBMS can periodically takes a *checkpoint* where it flushes all buffers out to disk.

How often the DBMS should take a checkpoint depends on the application's performance and downtime requirements. Taking a checkpoint too often causes the DBMS's runtime performance to degrade. But waiting a long time between checkpoints can potentially be just as bad, as the system's recovery time after a restart increases.

Blocking Checkpoint Implementation:

- The DBMS stops accepting new transactions and waits for all active transactions to complete.
- Flush all log records and dirty blocks currently residing in main memory to stable storage.
- Write a `<CHECKPOINT>` entry to the log and flush to stable storage.

7 Logging Schemes

Physical Logging:

- Record the 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 single page. Requires less data written in each log record than physical logging. Difficult to implement recovery with logical logging if you have concurrent transactions in a non-deterministic concurrency control scheme.
- 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.
- Most commonly used approach.

Lecture #21: ARIES Database Crash Recovery Algorithms

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 ARIES

Algorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics. Developed at IBM research in early 1990s. Not all systems implement ARIES exactly as defined in the original paper, but they are similar enough.

Main ideas of 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

We need to extend the DBMS's log record format to include additional info. Every log record now includes a globally unique *log sequence number* (LSN). Various components in the system keep track of **LSNs** that pertain to them:

- Each data page contains a *pageLSN*: The LSN of the most recent update to that page.
- System keeps track of *flushedLSN*: The max *LSN* flushed so far
- Before page i can be written to disk, we must flush log at least to the point where $pageLSN_i \leq flushedLSN$

3 Normal Execution

We first discuss the steps that the DBMS takes at runtime while it executes transactions.

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.

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.

We need to add another field to our log records called the *prevLSN*. This corresponds to the previous LSN for the transaction. The DBMS uses these *prevLSNs* to maintain a linked-list for each transaction that makes it easier to walk through the log to find its records.

We also need to introduce a new type of record called the *compensation log record* (CLR). A CLR describes the 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.

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.

We first discuss two blocking checkpoint methods where the DBMS pauses transactions during the checkpoint process. This pausing is necessary to ensure that the DBMS does not miss updates to pages during the checkpoint. We then present 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.

Blocking Checkpoints

The DBMS halts everything 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 on 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. We have to now record 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 (**R**unning, **C**ommitting, **U**ndo candidate)
- *lastLSN*: Most recent LSN written by transaction

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).

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.
- `<CHECKPOINT-END>`: When the checkpoint has completed. It contains the ATT + DPT.

5 ARIES Recovery

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

1. **Analysis**: Read the WAL to identify dirty pages in the buffer pool and active transactions at the time of the crash.
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.

Analysis Phase

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

- Scan log forward from the checkpoint.
- If the DBMS finds a `TXN-END` record, remove its transaction from ATT.
- All other records, add transaction to ATT with status **UNDO**, and on commit, change transaction status to **COMMIT**.
- 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 is to repeat history to reconstruct state at the moment of the crash. 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 greater than smallest *recLSN*, 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 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Distributed DBMSs

One can use the building blocks from single-node DBMSs to support transaction processing and query execution in distributed environments. An important goal in designing a distributed DBMS is to make it fault tolerant (i.e., to avoid 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

CPUs have access to common memory address space via a fast interconnect. CPUs also share the same disk. In practice, no one does this, as this is provided at the OS / kernel level. It causes problems, since each process's scope of memory is the same memory address space, which could 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.

Shared Disk

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

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

Some design questions to consider that will be covered more in-depth in future lectures:

- 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 between **homogeneous** and **heterogeneous** nodes, both used in modern-day systems:

Homogeneous: 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 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

Split database across multiple resources, including disks, nodes, processors. This process is sometimes called *sharding* in NoSQL systems. The DBMS executes query fragments on each partition and then combines the results to produce a single answer. Users should not be required to know where data is physically located and how tables are partitioned or replicated. A SQL query that works on a single-node DBMS should work the same on a distributed DBMS.

We want to pick a partitioning scheme that maximizes 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

Naive Table Partitioning: Each node stores one table, assuming enough storage space for a given node. This is each to implement as 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.

Horizontal Partitioning: Split 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 physical (shared nothing) or logically (shared disk) via hash partitioning or range partitioning.

5 Distributed Concurrency Control

If the DBMS supports multi-operation and distributed transactions, we need a way to coordinate their execution in the system.

Coordinator

This is a centralized approach with a global "traffic cop" that coordinates all the behavior. 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.

Middleware

This is the same as the centralized coordinator except that all queries are sent to a middleware directly layer.

Decentralized

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.