15445

wu

September 16, 2022

Contents

1 Introduction and the relational model **4**
1.1 Relational Algebra 5
1.2 Queries 8

2 Intermediate SQL **8**
2.1 Aggregates 9
2.2 Operations 11
2.2.1 String operations 11
2.3 Output 12
2.4 Nested Queries 13
2.5 Window Functions 14
2.6 Common table expressions 15

3 Database Storage **15**
3.1 File Storage 16
3.2 Page Layout 18
3.3 Tuple layout 19
3.4 Data representation 20
3.5 system catalogs 22
3.6 storage models 23

4 Buffer Pools **25**
4.1 Buffer Pool Manager 25
4.1.1 Multiple Buffer Pools 26
4.1.2 Pre-fetching 27
4.1.3 Scan Sharing 27
4.1.4 Buffer Pool Bypass 27
4.2 Replacement Policies 27

4.3 Other Memory Pools 28

5 Hashtables **28**
5.1 Hash functions 28
5.2 static hashing schemes 28
5.2.1 linear probe hashing 28
5.2.2 robin hood hashing 29
5.2.3 cuckoo hashing 29
5.3 dynamic hashing schemes 30
5.3.1 Chained hashing 30
5.3.2 extendible hashing 30
5.3.3 linear hashing 31

6 Tree Indexes **31**
6.1 B+ Tree overview 31
6.2 use in a DBMS 33
6.3 Design choices 33
6.3.1 node size 33
6.3.2 merge threshold 33
6.3.3 variable-length keys 33
6.3.4 intra-node search 34
6.4 optimizations 34
6.4.1 prefix compression 34
6.4.2 deduplication 34
6.4.3 bulk insert 34

7 Index Concurrency **34**
7.1 Latches Overview 34
7.1.1 Latch Modes 35
7.1.2 Latch Implementations 35
7.2 Hash table latching 36
7.3 B+Tree Latching 37
7.3.1 Latch crabbing/coupling 37
7.3.2 Better latching algorithm 38
7.4 Leaf Node Scans 38

8 Sorting & Aggregations **38**
8.1 External Mergre Sort 38
8.1.1 2-way external merge sort 39
8.1.2 General external merge sort 40

8.1.3 Using B+Trees for sorting . . . 40
8.2 Aggregations . . . 40
8.2.1 External hashing aggregate . . . 41
8.2.2 Hashing summarization . . . 42

9 Joins **42**
9.1 Join algorithms . . . 44
9.1.1 Nested Loop Join . . . 44
9.1.2 Sort-Merge Join . . . 45
9.1.3 Hash Join . . . 45

10 Query execution 1 **47**
10.1 Processing Models . . . 47
10.1.1 Iterator Model . . . 47
10.1.2 Materialization Model . . . 48
10.1.3 Vectorized/Batch Model . . . 49
10.2 Access Methods . . . 50
10.2.1 Sequential scan . . . 50
10.2.2 Index scan . . . 51
10.3 Modification Queries . . . 52
10.4 Expression Evaluation . . . 52

11 Query Execution 2 **53**
11.1 Process Models . . . 54
11.1.1 intra-operator (horizontal) . . . 56
11.1.2 inter-operator (vertical) . . . 56
11.1.3 bushy . . . 56
11.2 Execution Parallelism . . . 58
11.2.1 I/O parallelism . . . 58
11.3 I/O Parallelism . . . 59

12 Optimization 1 **59**
12.1 Relational Algebra Equivalences . . . 60
12.2 Logical Query Optimization . . . 61
12.2.1 Split Conjunctive Predicates . . . 61
12.2.2 Predicate Pushdown . . . 61
12.2.3 Replace Cartesian Products with Joins . . . 61
12.2.4 Projection Pushdown . . . 61
12.3 Nested Queries . . . 63
12.3.1 Rewrite . . . 63

12.3.2 Decompose 63
12.4 Expression Rewriting 64
12.5 Cost Model 64
12.6 More cost estimation 65
12.7 plan enumeration 69
12.7.1 single relation 69
12.7.2 multi-relation 70

13 Concurrency Control **70**
13.1 Atomicity 72
13.2 Consistency 72
13.3 Isolation 72
13.4 Durability 78

14 Two-Phase Locking **78**
14.1 Lock Types 78
14.2 Two-Phase Locking 79
14.3 Deadlock Detection + Prevention 82
14.3.1 Deadlock detection 82
14.3.2 Deadlock prevention 82
14.4 Hierarchical Locking 83
14.5 Conclusion 86

15 Timestamp Ordering Concurrency Control **86**
15.1 Basic Timestamp Ordering (T/O) protocol 87
15.2 Optimistic Concurrency Control 87
15.3 Isolation Levels 87

16 Lab notes **87**
16.1 project 3 87

17 Homework **88**
17.1 3 88
17.2 4 88

Query Planning
Operator Execution
Access Methods
Buffer Pool Manager
Disk Manager

1 Introduction and the relational model

Data model:

relational	most dbms
key/value graph document column family	NoSQL
Array/Matrix	Machine Learning
Hierarchical Network Multi-value	Obsolete/Legacy

The special value `NULL` is a member of every domain

A relation's **primary key** uniquely identifies a single tuple. Some DBMSs automatically create an internal primary key if a table does not define one.

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

Method to store and retrieve information from a database:

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

1.1 Relational Algebra

Select: $\sigma_{\text{predicate}}(R)$

Consider R($a_{id,bid}$)

a_{id}	b_{id}
a1	101
a2	102
a2	103
a3	104

By $\sigma_{\text{a\_id='a2'}}(R)$ we get

a_{id}	b_{id}
a2	102
a3	103

By $\sigma_{\text{a\_id='a2'} \wedge \text{b\_id>102}}(R)$ we get

a_{id}	b_{id}
a2	103

```
SELECT * FROM R
  WHERE a_id='a2' AND b_id>102;
```

Projection: $\Pi_{A1,\ldots,A_n}(R)$
By $\Pi_{\text{b\_id-100,a\_id}}(\sigma_{\text{a\_id='a2'}}(R))$ we get

b_{id}-100	a_{id}
2	a2
3	a2

```
SELECT b_id-100, a_id
    FROM R WHERE a_id='a2';
```

Union: $(R \cup S)$
Given $R(a_{id,bid})$

a_{id}	b_{id}
a1	101
a2	102
a3	103

and $S(a_{id,bid})$

a_{id}	b_{id}
a3	103
a4	104
a5	105

By $(R \cup S)$ we get

a_{id}	b_{id}
a1	101
a2	102
a3	103
a3	103
a4	104
a5	105

```
(SELECT * FROM R)
    UNION ALL
(SELECT * FROM S);
```

Intersection: $(R \cap S)$
By $(R \cap S)$ we get

a_{id}	b_{id}
a3	103

```
(SELECT * FROM R)
    INTERSECT
(SELECT * FROM S);
```

Difference: $(R - S)$ By $(R - S)$ we get

a_{id}	b_{id}
a1	101
a2	102

```
(SELECT * FROM R)
    EXCEPT
(SELECT * FROM S);
```

Product: $(R \times S)$
By $(R \times S)$ we get

R.a_{id}	R.b_{id}	S.a_{id}	S.b_{id}
a1	101	a3	103
a1	101	a4	104
a1	101	a5	105
a2	102	a3	103
a2	102	a4	104
a2	102	a5	105
a3	103	a3	103
a3	103	a4	104
a3	103	a5	105

```
SELECT * FROM R CROSS JOIN S;

SELECT * FROM R,S;
```

Join: $(R \bowtie S)$, generate a relation that contains all tuples that are a combination of two tuples with a common values for one or more attributes

By $(R \bowtie S)$ we get

a_{id}	b_{id}
a3	103

```
SELECT * FROM R NATURAL JOIN S;
```

Extra operators:

rename	ρ
assignment	$R \leftarrow S$
duplicate elimination	δ
aggregation	γ
sorting	τ
division	$R \div S$

1.2 Queries

The relational model is independent of any query language implementation

SQL is the standard

2 Intermediate SQL

Data Manipulation Language (DML)

Data Definition Language (DDL)

Data Control Language (DCL)

SQL is based on bags (duplicates) not sets (no duplicates)

Example database

student($_{\text{sid}}$, name, login, gpa)

sid	name	login	age	gpa
53666	Kanye	kanye@cs	44	4.0
53688	Bieber	jbieber@cs	27	3.9
53655	Tupac	shakur@cs	25	3.5

course($_{\text{cid,name}}$)

cid	name
15-445	Database Systems
15-721	Advanced Database Systems
15-826	Data Mining
15-823	Advanced Topics in Databases

enrolled($_{\text{sid}}$, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53655	15-445	B
63666	15-721	C

The basic syntax for a query is

```
SELECT column1, column2, ...
FROM table
WHERE predicate1, predicate2, ...
```

which students got an A in 15-721?

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

2.1 Aggregates

Functions that return a single value from a bag of tuples

- `AVG(col)` return the average col value
- `MIN(col)` return minimum col value
- `MAX(col)` return maximum col value
- `SUM(col)` return sum of values in col
- `COUNT(col)` return # of values for col

Aggregate functions can (almost) only be used in the `SELECT` output list
Get # of students with a "@cs" login:

```
SELECT COUNT(login) AS cnt
FROM student WHERE login LIKE '%@cs'
```

Get the number of students and their average GPA that have a "@cs" login

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

`COUNT`, `SUM`, `AVG` support `DISTINCT`
Get the number of unique students that have an "@cs" login

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

Output of other columns outside of an aggregate is undefined

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

Group by: Project tuples into subsets and calculate aggregates against each subset

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

From

e.sid	s.sid	s.gpa	e.cid
53435	53435	2.25	15-721
53439	53439	2.70	15-721
56023	56023	2.75	15-826
59439	59439	3.90	15-826
53961	53961	3.50	15-826
58345	58345	1.89	15-445

we get

AVG(s.gpa)	e.cid
2.46	15-721
3.39	15-826
1.89	15-445

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

```
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;
```

2.2 Operations

2.2.1 String operations

	String Case	String Quotes
SQL-92	Sensitive	Single Only
Postgres	Sensitive	Single Only
MySQL	Insensitive	Single/Double
SQLite	Sensitive	Single/Double
DB2	Sensitive	Single Only
Oracle	Sensitive	Single Only

```
WHERE UPPER(name) = UPPER('KaNyE') /*SQL-92*/

WHERE name = "KaNyE"               /*MySQL*/
```

LIKE is used for string matching
'%' matches any substring, '_' matches any one character

```
SELECT SUBSTRING(name, 1, 5) AS abbrv_name
FROM student WHERE sid = 53688

SELECT * FROM student AS s
WHERE UPPER(s.name) LIKE 'KAN%'
```

SQL standard says to use || operator to concatenate two or more strings together, MySQL uses +
DATE/TIME

```
SELECT NOW();

SELECT CURRENT_TIMESTAMP;

SELECT EXTRACT(DAY FROM DATE('2021-09-01'));

SELECT DATE('2021-09-01') - DATE('2021-01-01') AS days;

SELECT ROUND((UNIX_TIMESTAMP(DATE('2021-09-01')) - UNIX_TIMESTAMP(DATE('2021-01-01'))

SELECT DATADIFF(DATE('2021-09-01'), DATE('2021-01-01')) AS days;

SELECT juliaday(CURRENT_TIMESTAMP) - julianday('2021-01-01');
```

```
SELECT CAST((julianday(CURRENT_TIMESTAMP) - julianday('2021-01-01')) AS INT) AS days;
```

2.3 Output

Store query results in another table

- table must not already be defined
- table will have the same # of columns with the same types as the input

```
SELECT DISTINCT cid INTO CourseIds
FROM enrolled;                    /*SQL-92*/

CREATE TABLE CourseIds (
SELECT DISTINCT cid FROM enrolled);/*MySQL*/
```

Insert tuples from query into another table

- Inner SELECT must generate the same columns as the target table
- DBMSs have the different options/syntax on what to do with integrity violations

```
INSERT INTO CourseIds
(SELECT DISTINCT cid FROM enrolled); /*SQL-92*/
```

ORDER BY <column*> [ASC|DESC]

- Order the output tuples by the vgalues in one or more of their columns

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

LIMIT <count> [offset]

- limit the # of tuples returned in output
- Can set an offset to return a "range"

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

2.4 Nested Queries

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

Get the names of students in '15-445'

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

- `ALL`: must satisfy expression for all rows in the sub-query
- `ANY`: must satisfy expression for at least one row in the sub-query
- `IN`: equivalent to '`=ANY()`'
- `EXISTS`: at least one row is returned

Get the names of students in '15-445'

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

Find student record with the highest id that is enrolled in at least one course

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

SELECT sid,name FROM student
WHERE sid IN (
  SELECT MAX(sid) FROM enrolled
  ORDER BY sid DESC LIMIT 1
);
```

Find all courses that have no students enrolled in it

```
SELECT * FROM course
WHERE NOT EXISTS (
  SELECT * FROM enrolled
  WHERE course.cid = enrolled.cid
)
```

PROBLEM

2.5 Window Functions

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

Special windows functions

- `ROW_NUMBER()` - # of the current window
- `RANK()` - Order positions of the curfrent row

```
SELECT *, ROW_NUMBER() OVER() AS row_num
FROM enrolled
```

sid	cid	grade	row_{num}
53666	15-445	C	1
53688	15-721	A	2
53688	15-826	B	3
53655	15-445	B	4
53666	15-721	C	5

The `OVER` keyword 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
```

cid	sid	$\text{row}_{\text{number}}$
15-445	53666	1
15-445	53655	2
15-721	53688	1
15-721	53666	2
15-826	53688	1

You can also include an `ORDER BY` in the window grouping to sort entries in each group.

Find the student with the second highest grade for each course

```
SELECT * FROM (
  SELECT *, RANK() OVER (PARTITION BY cid ORDER BY grade ASC) AS rank
  FROM enrolled
) AS ranking
WHERE ranking.rank = 2
```

2.6 Common table expressions

Provides a way to write auxiliary statements for use in a larger query

```
WITH cteSource(maxID) AS (
  SELECT MAX(sid) FROM enrolled
)
SELECT name FROM student, cteSource
WHERE student.sid = cteSource.maxId
```

Print the sequence of numbers from 1 to 10

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

3 Database Storage

- `madvice`: tell the os how you expect to read certain pages
- `mlock`: tell the os that memory ranges cannot be paged out
- `msync`: tell the os to flush memory ranges out to disk

DBMS (almost) always wants to control things itself and can do a better job than the OS

Problem 1: How the DBMS represents the database in files on disk

Problem 2: How the DBMS manages its memory and moves data back-and-forth from disk

3.1 File Storage

The **storage manager** is responsible for maintaining a database's files
It organizes the files as a collection of **pages**

- tracks data read/written to pages
- tracks the available space

A **page** is a fixed-size block of data
Each page is given a unique identifier

- The DBMS uses an indirection layer to map page IDs to physical locations

There are three different notions of "pages" in a DBMS:

- Hardware Page (4KB)
- OS Page (4KB)
- Database Page (512B-16KB)

A hardware page is the largest block of data that the storage device can guarantee failsafe writes
A **heap file** is an unordered collection of pages with tuples that are stored in random order

- create/get/write/delete page
-

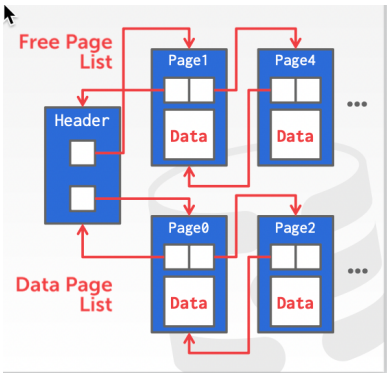
Two ways to represent a heap file

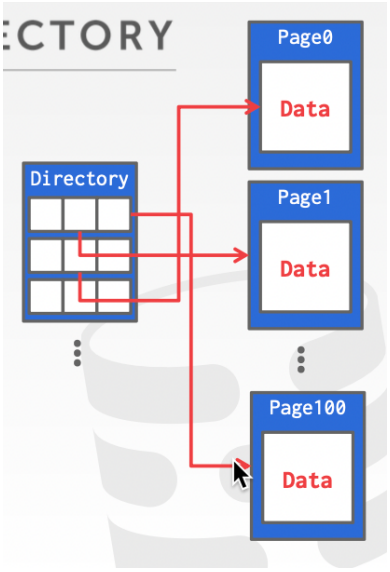
- linked list
- page directory

Linked List: maintain a **header page** at the beginning of the file that stores two pointers

- HEAD of the **free page list**
- HEAD of the **data page list**

Each page keeps track of how many free slots they currently have
The DBMS maintains special pages that tracks the location of data pages in the database files
The directory also records the number of free slots per page
must make sure that the directory pages are in sync with the data pages


Free Page List
Header
Page1
Data
Page4
Data
Page0
Data
Page2
Data
Data Page List


ECTORY
Directory
Page0
Data
Page1
Data
Page100
Data

3.2 Page Layout

Every page contains a **header** of metadata about the page's content

- page size
- checksum
- DBMS version
- transaction visibility
- compression information

Some systems require pages to be self-contained

For any page storage architecture, we need to decide how to organize the data inside of the page

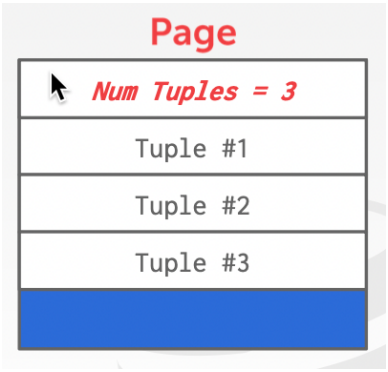
Two approaches

- tuple-oriented
- log-structured

Tuple-oriented:

Strawman Idea: keep track of the number of tuples in a page and then just append a new tuple to the end

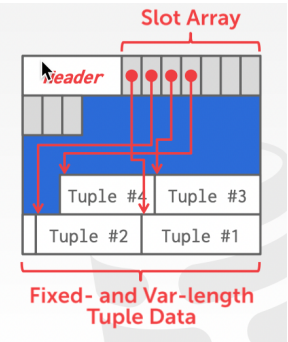
- What happens if we delete a tuple
- what happens if we have a variable-length attribute



The most common layout scheme is called **slotted pages**, the slot array maps "slots" to the tuples' starting position offsets

The header keeps track of

- the # of used slots
- The offset of the starting location of the last slot used



The DBMS needs a way to keep track of individual tuples, each tuple is assigned a unique **record identifier**

- most common: `page_id` + `offset/slot`

An application cannot rely on these IDs to mean anything

3.3 Tuple layout

A tuple is essentially a sequence of bytes

It's the job of the DBMS to interpret those bytes into attribute types and values

Each tuple is prefixed with a **header** that contains meta-data about it

- visibility info
- bit map for `NULL` values

We do **not** need to store meta-data about the schema

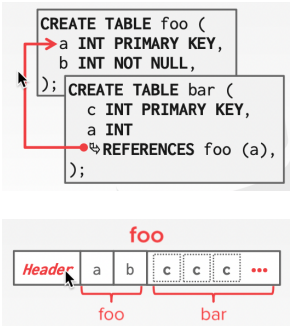
Attributes are typically stored in the order that you specify them when you create the table.

DBMS can physically **denormalize** (pre join) related tuples and store them together in the same page

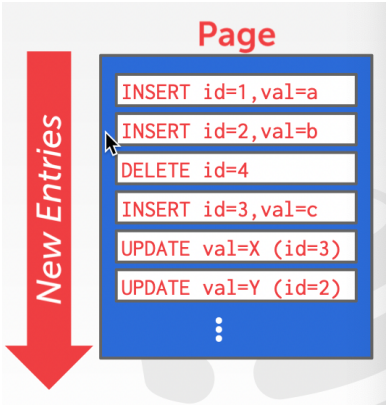
Instead of storing tuples in pages, the DBMS only stores **log records**

The system appends log records to the file of how the database was modified

- inserts store the entire tuple



- deletes mark the tuple as deleted
- updates contain the delta of just the attributes that were modified



To read as records, the DBMS scans the log backwards and "recreates" the tuple to find what it needs

Build indexes to allow it to jump to locations in the log

Periodically compact the log

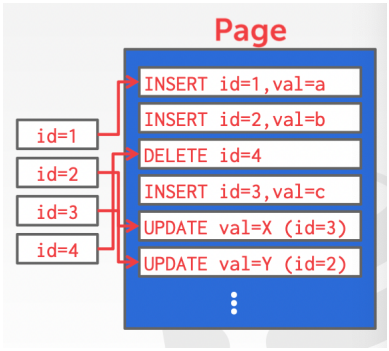
Compaction coalesces larger log files into smaller files by removing unnecessary records

3.4 Data representation

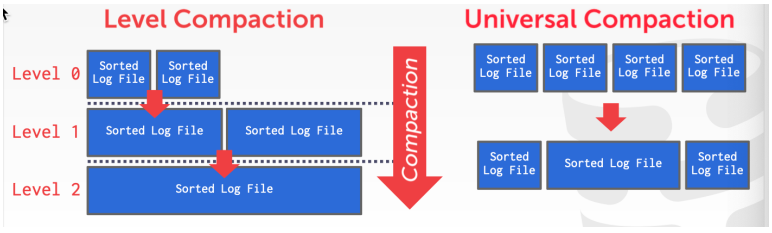
- `INTEGER` / `BIGINT` / `SMALLINT` / `TINYINT` C/C++ Representation
- `FLOAT` / `REAL` vs. `NUMERIC` / `DECIMAL`

  IEEE-754 Standard / Fixed-point Decimals

  numerical/decimal is accurate without rounding errors


Page
INSERT id=1,val=a
INSERT id=2,val=b
DELETE id=4
INSERT id=3,val=c
UPDATE val=X (id=3)
UPDATE val=Y (id=2)
id=1
id=2
id=3
id=4


Page
id=1,val=a
id=2,val=Y
id=3,val=X
DELETE id=4


Level Compaction
Universal Compaction
Level 0
Level 1
Level 2
Sorted Log File
Compaction

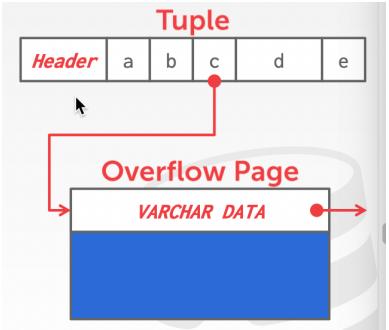
- `VARCHAR` / `VARBINARY` / `TEXT` / `BLOB`

  Header with length, followed by data bytes.

- `TIME` / `DATE` / `TIMESTAMP`

  32/64-bit integer of (micro)seconds since Unix epoch

To store values that are larger than a page, the DBMS uses separate **overflow** storage pages



Some systems allow you to store a really large value in an external file, treated as a `BLOB` type

The DBMS **cannot** manipulate the contents of an external file

3.5 system catalogs

A DBMS stores meta-data about databases in its internal catalogs

- tables, columns, indexes, views
- users, permissions
- internal statistics

Almost every DBMS stores the database's catalog inside itself

You can query the DBMS's internal `INFORMATION_SCHEMA` catalog to get info about the database

List all the tables in the current database:

```
/*SQL-92*/
SELECT *
FROM INFORMATION_SCHEMA.TABLES
WHERE table_catalog = '<db_name>';
```

```
/*Postgres*/
\d;

/*MySQL*/
SHOW TABLES;

/*SQLite*/
.tables
```

List all the tables in the student table

```
/*SQL-92*/
SELECT *
FROM INFORMATION_SCHEMA.TABLES
WHERE table_catalog = 'student';

/*Postgres*/
\dstudent;

/*MySQL*/
DESCRIBE student;

/*SQLite*/
.schema student
```

Database workloads:

- On-line transaction processing (OLTP)

  fast operations that only read/update a small amount of data each time

- On-line analytical processing (OLAP)

  complex queries that read a lot of data to compute aggregates

- Hybrid transaction + analytical processing (HTAP)

  OLTP+OLAP together on the same database instancew

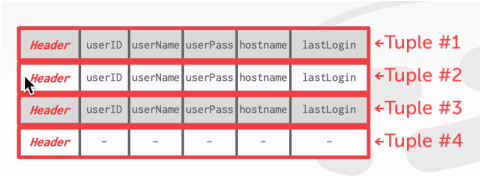
3.6 storage models

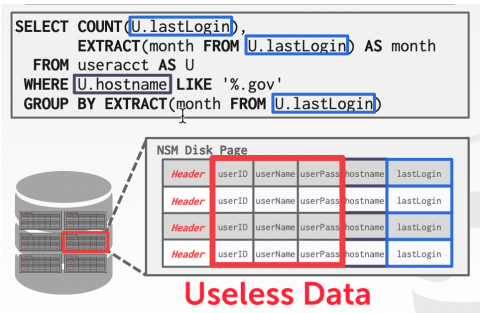
The DBMS cna store tuples in different ways that are better for either OLTP or OLAP workloads

We haven been assuming the n-ary storage model so fart this semester

n-ary storage model (NSM): the DBMS stores all attributes for a single tuple contiguously in a page

Ideal for OLTP workloads where queries tend to operate only on an individual entity and insert-heavy workloads





Advantages

- fast insertions, updates and deletes
- Good for querries that need the entire tuple

Disadvantages

- not good for scanning large portions of the table and/or a subset of the attributes

decomposition storage model (DSM): the DBMS stores the values of a single attribute for all tupoles contiguously in a page

- also known as a "column store"

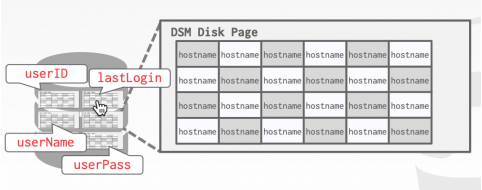
Tuple identification:

- fixed-length offsets

  each value is the same length for an atrribute

- embedded tuple Ids

  each value is stored with its tuple id in a column



4 Buffer Pools

How the DBMS manages its memory and move data back-and-forth from disk

- spatial control
  - where to write pages on disk
  - the goal is to keep pages that are used together often as physically close together as possible on disk
- temporal control
  - when to read pages into memory, and when to write them to disk
  - the goal is to minimize the number of stalls from having to read data from disk

4.1 Buffer Pool Manager

Memory region organized as an array of fixed-size pages. An array entry is called a **frame**

When the DBMS requests a page, an exact copy is places into one of these frames

The **page table** keeps track of pages that are currently in memory
Also maintains additional meta-data per page

- dirty flag
- pin/reference counter

Locks

- protects the database's logical contents 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 structure from other threads
- held for operation duration
- do not need to be able to rollback changes

The **page directory** is the mapping from page ids to page locations in the database files

- all changes must be recorded on disk to allow the DBMS to find on restart

The **page table** is the mapping from page ids to a copy of the page in buffer pool frames

- this is an in-memory data structure that does not need to be stored on disk

Buffer pool optimizations

- multiple buffer pools
- pre-fetching
- scan sharing
- buffer pool bypass

4.1.1 Multiple Buffer Pools

The DBMS does not always have a single buffer pool for the entire system

- multiple buffer pool instances
- per-database buffer pool
- per-page type buffer pool

Helps reduce latch contention and improve locality
Approach 1: Object Id

- Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools

Approach 2: Hashing

- Hash the page id to select which buffer pool to access

4.1.2 Pre-fetching

The DBMS can also prefetch pages based on query plan

4.1.3 Scan Sharing

Queries can reuse data retrieved from storage or operator computations

- Also called **synchronized scans**

Allow multiple queries to attach to a single cursor that scans a table

- queries don't have to be the same
- can also share intermediate results

4.1.4 Buffer Pool Bypass

The sequential scan operator won't store fetched pages in the buffer pool to avoid overhead

4.2 Replacement Policies

Least-recently used

Approximation of LRU that does not need 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 page's bit is set to 1
- if yes, set to zero. If no, then evict

Better policies:

- LRU-K

  Track the history of last K references to each page as timestamps and compute the interval between subsequent accesses

  The DBMS then uses this history to estimate the next time that page is going to be accessed

- The DBMS chooses which pages to evict on a per txn/query basis.

4.3 Other Memory Pools

- sorting + join buffers
- query caches
- maintenance buffers
- log buffers
- dictionary caches

5 Hashtables

We are now going to talk about how to support the DBMS's execution engine to read/write data from pages

5.1 Hash functions

- crc-64 (1975)
- murmurhash (2008)
- google cityhash (2011)
- facebook xxhash (2012)
- google farmhash (2014)

5.2 static hashing schemes

5.2.1 linear probe hashing

single giant table of slots
resolve collisions by linearly searching for the next free slot in the table

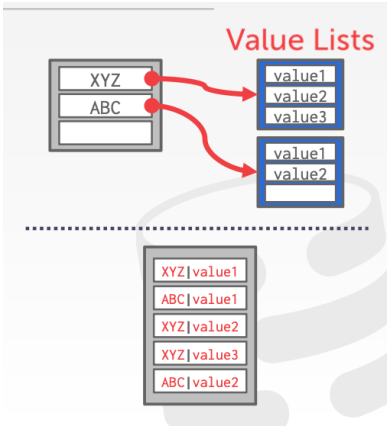
- to determine whether an element is present, hash to a location in the index and scan for it
- must store the key in the index to know when to stop scanning
- insertions and deletions are generalizations of lookups

delete: support A and B are hashed into the same location and then B is the next element of A, now if we delete A, how do we find the B

- tombstone
- movement

For non-unique keys,

1. separated linked list
2. redundant keys



5.2.2 robin hood hashing

Variant of linear probe hashing that steals 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.

5.2.3 cuckoo hashing

Use multiple hash tables with different hash functions seeds

- on insert, check every table and pick anyone that has a free slot
- if no table has a free slot, evict the element from one of them and then re-hash it find a new location

Look-ups and deletions are always O(1) because only one location per hash table is checked

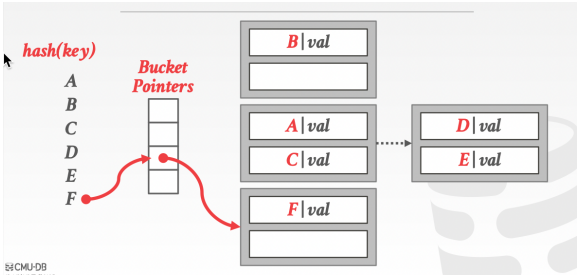
5.3 dynamic hashing schemes

5.3.1 Chained hashing

maintain a linked list of **buckets** for each slot in the hash table

resolve collisions by replacing all elements with the same hash key into the same bucket

- to determine whether an element is present, hash to its buckets and scan for it



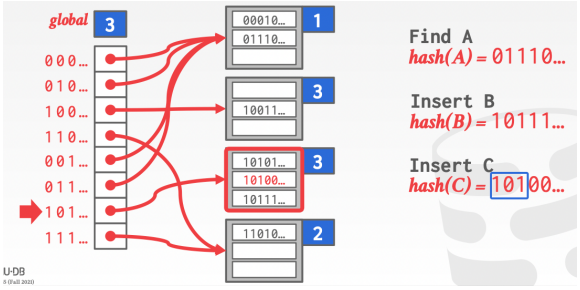
5.3.2 extendible hashing

better source

chained-hashing approach where we split buckets instead of letting the linked list grow forever

multiple slot locations can point to the same bucket chain

reshuffle bucket entires on split and increase the number of bits to examine



5.3.3 linear hashing

The hash table maintains a **pointer** that tracks the next bucket to split

- when any bucket overflows, split the bucket at the pointer location

use multiple hashes to find the right bucket for a given key
can use different overflow criterion

6 Tree Indexes

A **table index** is a replica of a subset of a table's attributes that are organized and/or sorted for efficient using those attributes

6.1 B+ Tree overview

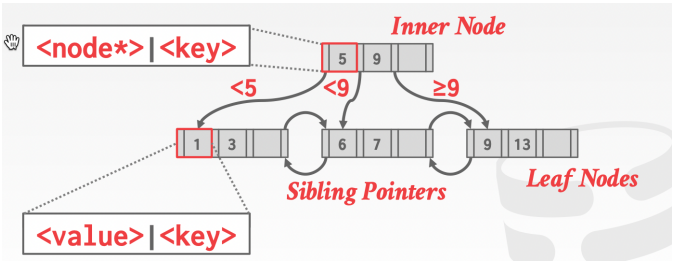
B-tree, B+tree, B*tree, Blink-tree

A B+Tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions and deletions in $O(\log n)$

- optimized for systems that read and write large blocks of data

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 in the tree)
- every node other than the root is at least half-full $M/2 - 1 \leq$ #keys $\leq M - 1$
- every inner node with k keys has $k + 1$ non-null children



Every B+Tree node is comprised of an array of key/value pairs

- the keys are derived from the attributes that the index is based on

- the values will differ based on whether the node is classified as an **inner node** or a **leaf node**

The arrays are (usually) kept in sorted key order
Leaf node values approach

1. record IDs

   A pointer to the location of the tuple to which the index corresponds

2. tuple data

   the leaf nodes store the actual contents of the tuple

   secondary indexes must store the record ID as their values

Insert

1. find correct leaf node L
2. put data entry into L in sorted order
3. if L has enough space, done
4. otherwise, split L keys into L and a new node L2
   - redistribute entries evenly, copy up middle key
   - insert index entry pointing to L2 into parent of L

Delete

1. find leaf L where entry belongs. remove the entry
2. if L is at least half-full, done
3. if L has only M/2-1 entries
   - try to re-distribute, borrowing from sibling
   - if re-distribution fails, merge L and sibling

If merge occured, must delete entry from parent of L
Duplicate keys

1. append record ID
   - add the tuple's unique record ID as part of the key to ensure that all keys are unique

- the DBMS can still use partial keys to find the tuples

2. Overflow leaf nodes
    - allow leaf nodes to spill into overflow nodes that contain the duplicate keys

clustered indexes The table is stored in the sort order specified by the primary key

- can be either heap- or index-organized storage

some DBMS always use a clustered index

- if a table does not contain a primary key, the DBMS will automaticall y make a hidden primary key

6.2 use in a DBMS

6.3 Design choices

6.3.1 node size

the slower the storage device, the larger the optimal node size for a B+ Tree

- HDD: 1MB
- SSD: 10KB
- In-Memory: 512B

optimal sizes can vary depending on the workload

6.3.2 merge threshold

some DBMSs do not always merge nodes when they are half full
delaying a merge operation may reduce the amount of reorganization
it may also be better to just let smaller nodes exist and then periodically rebuild entire tree

6.3.3 variable-length keys

1. pointers
2. variable-length nodes
3. padding
4. key map / indirection

6.3.4 intra-node search

1. linear
2. binary
3. interpolation

6.4 optimizations

6.4.1 prefix compression

sorted keys in the smae leaf node are likely to have the same prefix

robbed robbing robot

Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key

6.4.2 deduplication

non-unique indexes can end up storing multiple copies of the same key in leaf nodes

the leaf node can store the key once and then maintain a list of tuples with that key

6.4.3 bulk insert

The fastest way to build a new B+Tree for an existing table is to first sort the keys and then rebuild the index from the bottom up

7 Index Concurrency

7.1 Latches Overview

Locks

- protect the database's logical contents from other txns
- held for txn duration
- need to be able to rollback changes

Latches

- Protect the critical sections of the DBMS's internal data structure from other threads
- held for operation duration
- do not need to be able to rollback changes

	Locks	Latches
Separate	User Txns	Threads
Protect	Database Contents	In-Memory Data Structures
During	Entire Txns	Critical Sections
Modes	Shared, Exclusive, Update, Intention	Read, Write
Deadlock	Detection & Resolution	Avoidance
by	Waits-for, Timeout, Aborts	Coding Discipline
Kept in	Lock Manager	Protected Data Structure

7.1.1 Latch Modes

Read Mode

- Multiple threads can read the same object at the same time
- A thread can acquire the read latch if another thread has it in read mode

Write Mode

- Only one thread an access the object
- A thread cannot acquire a write latch if another thread has it in any mode

7.1.2 Latch Implementations

1. Blocking OS Mutex non-scalable (about 25ns per lock/unlock invocation)

```
std::mutex m;

m.lock();

m.unlock();
```

But `std::mutex` -> `pthread_mutex_t` -> `futex`

2. Test-and-Set Spin Latch (TAS)

   - very efficient (single instruction to latch/unlatch)
   - non-scalable, not cache-friendly, not OS-friendly
   - `std::atomic<T>`

   ```
   std::atomic_flag latch;

   while (latch.test_and_set(...)) {

   }
   ```

   Do not use spinlocks in user space, unless you actually know what you're doing. And be aware that the likelihood that you know what you are doing is basically nil.

3. Read-Writer Latches

   - Allows for concurrent readers
   - Must manage read/write queues to avoid starvation
   - can be implemented on top of spin latches

7.2 Hash table latching

easy to support concurrent access due to the limited ways threads access the data structure

- all threads move in the same direction and only access a single page/slot at a time
- deadlocks are not possible

To resize the table, take a global write latch on the entire table

1. 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

2. Slot latches
    - each slot has its own latch
    - can use a single-mode latch to reduce meta-data and computational overhead

Atomic instruction that compares contents of a memory location M to a given value V `__sync_bool_compare_and_swap(&M,20,30)`

- if values are equal, installs new given value V′ in M
- otherwise operation fails

7.3 B+Tree Latching

We want to allow multiple threads to read and update a B+ Tree at the same time

We need to protect against two types of problems

- threads trying to modify the contents of a node at the same time
- one thread traversing the tree while another thread splits/merge nodes

7.3.1 Latch crabbing/coupling

Protocol to allow multiple threads to access/modify B+ Tree at the same time

Basic idea:

- get latch for parent
- get latch for child
- release latch for parent if "safe"

A **safe node** is one that will not split or merge when updated

- not full
- more than half-full

Find: start at root and go down

- acquire R latch on child

- then unlatch parent

Insert/Delete: Start at root and go down, obtaining W latches as needed. Once child is latched, check if it is safe:

- if child is safe, release all latches on ancestors

But taking a write latch on the root every time becomes a bottleneck with higher concurrency

7.3.2 Better latching algorithm

Most modifications to a B+Tree will not require a split or merge

Instead of assuming that there will be a split/merge, optimistically traverse the tree using read latches

If you guess wrong, repeat traversal with the pessimistic algorithm

Search: same as before

Insert/Delete:

- set latches as if for search, get to leaf, and set W latch on leaf
- if leaf is not safe, release all latches, and restart thread using previous insert/delete protocol with write latches

This approach optimistically assumes that only leaf node will be modified; if not, R latches set on the first pass to leaf are wasteful

7.4 Leaf Node Scans

The threads in all the examples so far have acquired latches in a "top-down" manner

But what if we want to move from one leaf node to another leaf node?

Latches do not support deadlock detection or avoidance. 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

The DBMS's data structures must cope with failed latch acquisitions

8 Sorting & Aggregations

8.1 External Mergre Sort

What do we need to sort

- relational model/SQL is unsorted
- queries may request that tuples are sorted in a specific way
- But even if a query does not specify an order, we may still want to sort to do other things
  - trivial to support duplicate elimination
  - bulk loading sorted tuples into a B+ tree index is faster
  - aggregations

8.1.1 2-way external merge sort

2 is the number of runs that we are going to merge into a new run for each pass

data is broken up into N pages

the DBMS has a finite number of B buffer pool pages to hold input and output data

Pass 0

- read all B pages of the table into memory
- sort pages into runs and write them back to disk

Pass 1,2,3,..

- recursively merge pairs of runs into runs twice as long
- uses three buffer pages (2 for input pages, 1 for output)

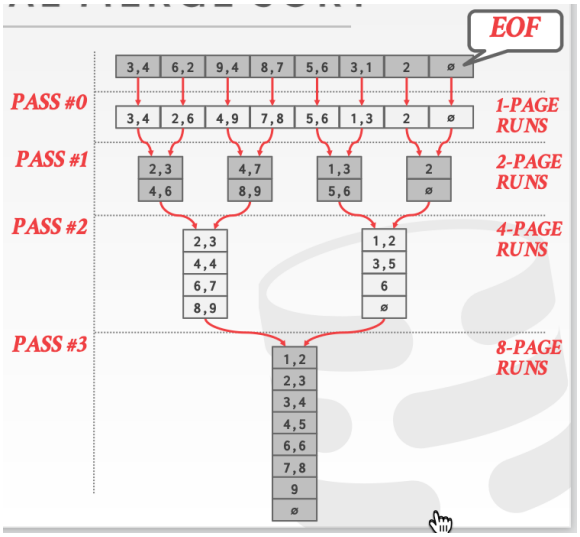
Number of pass: $1 + \lceil \log_2 N \rceil$

Total I/O cost: $2N \cdot (\#$ of passes$)$

This algorithm only requires three buffer pool pages to perform the sorting

Double buffering optimization Prefetch the next run in the background and store it in a second buffer while system is processing the current run

- reduces the wait time for I/O requests at each step



8.1.2 General external merge sort

Pass 0

- use B buffer pages
- produce $\lceil N/B \rceil$ sorted runs of size B

Pass 1

- merge $B - 1$ runs

Number of pass: $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
Total I/O cost: $2N \cdot (\#$ of passes$)$

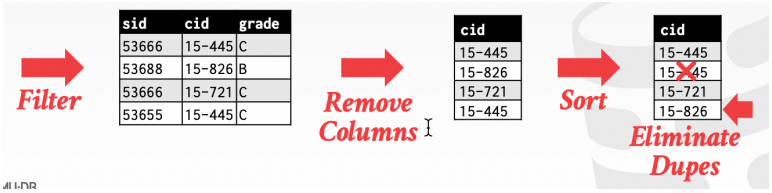
8.1.3 Using B+Trees for sorting

8.2 Aggregations

Two implementation choices

- sorting
- hashing

Hashing aggregate: Populate an ephemeral hash table as the DBMS scans the table. For each record, check whether there is already an entry in the hash table:



- `DISTINCT`: discard duplicate
- `GROUP BY`: perform aggregate computation

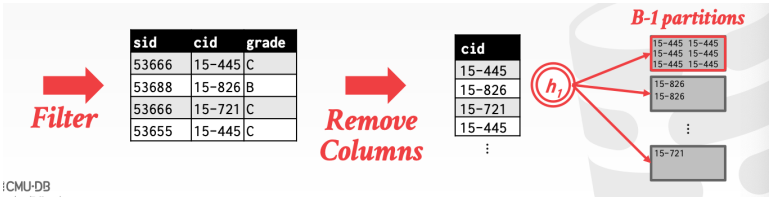
If everything fits in memory, then this is easy

8.2.1 External hashing aggregate

1. Phase 1: Partition
   - divide tuples into buckets based on hash key
   - write them out to disk when they get full

   use a hash function h_1 to split tuples into **partitions** on disk
   - a partition is one or more pages that contain the set of keys with the same hash value
   - partitions are "spilled" to disk via output buffers

   Assume that we have B buffers, we will use B-1 buffers for the partitions and 1 buffer for the input data



2. Phase 2: ReHash
   - build in-memory hash table for each partition and compute the aggregation

   For each partition on disk

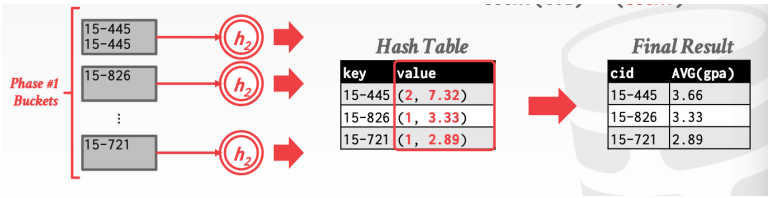
- read it into memory and build an in-memory hash table based on a second hash function h_2
- then go through each bucket of this hash table to bring together matching tuples

This assumes that each partition fits in memory

8.2.2 Hashing summarization

During the rehash phase, store pairs of the form `GroupKey->RunningVal` when we want to insert a new tuple into the hash table

- if we find a matching `GroupKey`, just update the `RunningVal` appropriately
- else insert a new `GroupKey->RunningVal`



9 Joins

We will focus on performing binary joins (two tables) using **inner equijoin** algorithms

- these algorithms can be tweaked to support other joins
- multi-way joins exist primarily in research literature

In general, we want the smaller table to always be the left table ("outer table") in the query plan

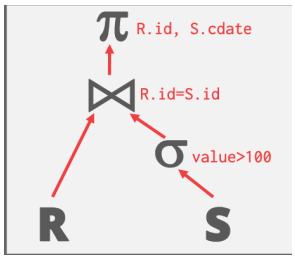
Decision 1: output

- what data does the join operator emit to its parent operator in the query plan tree

Decision 2: cost analysis criteria

- how do we determine whether one join algorithm is better than another

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



For tuple $r \in R$ and tuple $s \in S$ that match on join attributes, concatenate r and s together into a new tuple
output contents can vary:

- depends on processing model
- depends on storage model
- depends on data requirements in query

Early Materialization:

- copy the values for the attributes in outer and inner tuples into a new output tuple
- subsequent operators in the query plan never need to go back to the base tables to get more data

Late Materialization:

- only copy the joins keys along with the Record IDs of the matching tuples
- ideal for column stores because the DBMS does not copy data that is not needed for the query

Cost Analysis Criteria
Assume

- M pages in table R, m tuples in R
- N pages in table S, n tuples in S

Cost Metric: # of IOs to compute join

$R \bowtie S$ is the most common operation and thus must be carefully optimized

$R \times S$ followed by a selection is inefficient because the cross-product is large

9.1 Join algorithms

9.1.1 Nested Loop Join

1. Simple/Stupid foreach tuple $r \in R$: foreach tuple $s \in S$: emit, if r and s match

   Cost: $M + m \cdot N$

2. Block foreach block $B_R \in R$ foreach block $B_S \in S$ foreach tuple $r \in B_r$ foreach tuple $s \in B_s$ emit, if r and s match

   cost: $M + M \cdot N$

   What if we have B buffers available?

   - use $B - 2$ buffers for scanning the outer table
   - use one buffer for the inner table, one buffer for storing output

   foreach $B - 2$ blocks $b_R \in R$ foreach block $b_S \in S$ foreach tuple $r \in B - 2$ blocks foreach tuple $s \in b_S$ emit, if r and s match

   Cost: $M + \lceil M/(B-2) \rceil \cdot N$

3. Index Why is the basic nested loop join so bad?

   - for each tuple in the outer table, we must do a sequential scan to check for a match in the inner table

   We can avoid sequential scans by using an index to find inner table matches

   - use an existing index for the join

   foreach tuple $r \in R$ for each tuple $s \in$ Index$(r_i = s_j)$ emit, if r and s match

9.1.2 Sort-Merge Join

Phase 1: sort

- sort both tables on the join keys

Phase 2: merge

- step through the two sorted tables with cursors and emit matching tuples
- may need to backtrack depending on the join type

sort R, S on join keys cursor$_S \leftarrow R_{\text{sorted}}$, cursor$_S \leftarrow S_{\text{sorted}}$ while cursor$_R$ and cursor$_S$: if cursor$_R >$ cursor$_S$ increment cursor$_S$ if cursor$_R <$ cursor$_S$ increment cursor$_R$ elif cursor$_R$ and cursor$_S$ match: emit increment cursor$_S$

Sort Cost(R): $2M \cdot (1 + \lceil \log_{B-1} \lceil M/B \rceil \rceil)$ Sort Cost(S): $2N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$
Merge Cost: $M + N$

When is sort-merge join useful?

- one or both tables are already sorted on join key
- output must be sorted on join key
- the input relations may be sorted either by an explicit sort operator, or by scanning the relation using an index on the join key

9.1.3 Hash Join

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 partition i, the R tuple must be in r_i and the S tuple in s_i

Therefore R tuples in r_i need only to be compared with S tuples in s_i

Phase 1: build

- scan the outer relation and populate a hash table using the hash function h_1 on the join attributes

Phase 2: probe

- scan the inner relation and use h_1 on each tuple to jump to a location in the hash table and find a matching tuple

Hash table contents
key: the attributes
value: varies per implementation

- depends on what the operators above the join in the query plan expect as its input

Approach 1: full tuple
Approach 2: tuple identifier

- could be to either the base tables or the intermediate output from child operators in the query plan
- ideal for column stores because the DBMS does not fetch data from disk that it does not need
- also better if join selectivity is low

Probe phase optimization: create a **Bloom Filter** during the build phase when the key is likely to not exist in the hash table

- threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches
- sometimes called **sideways information passing**

Bloom filters is a probalistic data structure (bitmap) that answers set membership queries

- false negatives will never occur
- false positives can sometimes occur

`Insert(x)`: use k hash functions to set bits in the filter to 1
`Lookup(x)`: check whether the bits are 1 for each hash function
how big of a table can we hash using this approach?

- $B-1$ "spill partitions" in phase 1
- each should be no more than B blocks big

Answer: $B \cdot (B-1)$

- a table of N pages needs about $\sqrt{N}$ buffers
- assume hash distributes records evenly. Use a "fudge factor" $f > 1$ for that: we need $B \cdot \sqrt{fN}$

What happens if we do not have enough memory to fit the entire hash table?

we do not want to let the buffer pool manager swap out the hash table pages at random

Hash join when tables do not fit in memory

- Build Phase: Hash both tables on the join attribute into partitions
- Probe Phase: Compares tuples in corresponding partitions for each table

Cost: $3(M + N)$
partition: $2(M + N)$
probing $M + N$

algorithm	IO cost
simple nested loop join	$M + (m \cdot N)$
block nested loop join	$M + (M \cdot N)$
index nested loop join	$M + (M \cdot C)$
Sort-Merge join	$M + N$+sort cost
hash join	$3(M + N)$

10 Query execution 1

10.1 Processing Models

A DBMS's **processing model** defines how the system executes a query plan

- different trade-offs for different workloads

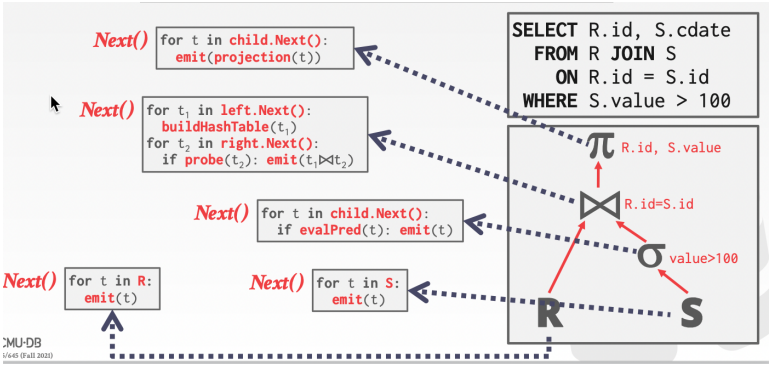
10.1.1 Iterator Model

Each query plan operator implements a `next()` function

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

Also called **volcano** or **pipeline** model

This is used in almost every DBMS. Allows for tuple **pipelining**
some operators must block until their children emit all their tuples



- joins, subqueries, order by

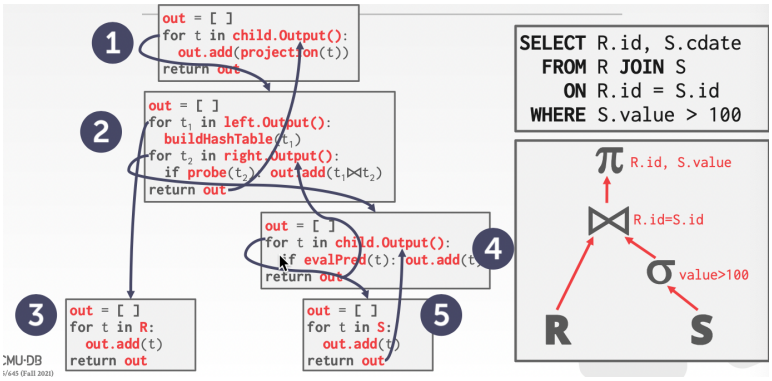
output control works easily with this approach

10.1.2 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
- the BDMS can push down hints (e.g. `LIMIT`) to avoid scanning too many tuples
- can send either a materialized row or a single column

The output can be either whole tuples (NSM) or subsets of columns (DSM)



better for OLTP workloads because queries only access a small number of tuples at a time

- lower execution / coordinaten overhead
- fewer function calls

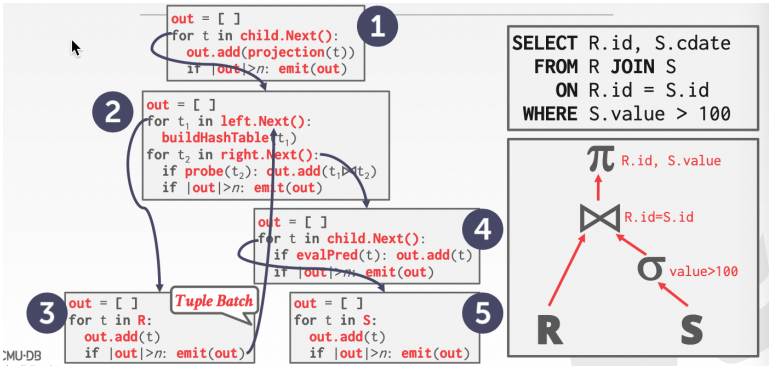
not good for OLAP queires with large intermediate results

10.1.3 Vectorized/Batch Model

like the iterator model where each operator implements a `next()` function, but

each operator emits a **batch** of tuples instead of single tuple

- the operator's internal loop processes multiple tuples at a time
- the size of the batch can vary based on hardware or query properties



Ideal for OLAP queries because it greatly reduces the number of invocations per operator

Allows for operators to more easily use vectorized (SIMD) instructions to process batches of tuples

Plan processing direction

- top-to-bottom
  - start with the root and "pull" data up from its children
  - tuples are always passed with function calls
- bottom-to-top
  - start with leaf nodes and push data to their parents
  - allows for tighter control of caches/registers in pipelines

10.2 Access Methods

An **access method** is the way that the DBMS accesses the data stored in a table

- not defined in relational algebra

10.2.1 Sequential scan

for each page in the table

- retrieve it from the buffer pool
- iterate over each tuple and check whether to include it

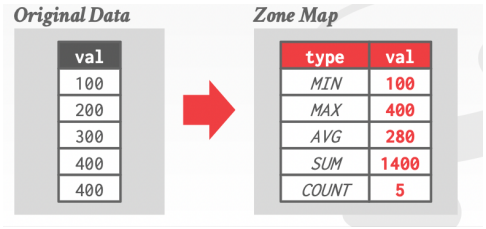
for page in table.pages: for t in page.tuples: if evalPred(t) // do something

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

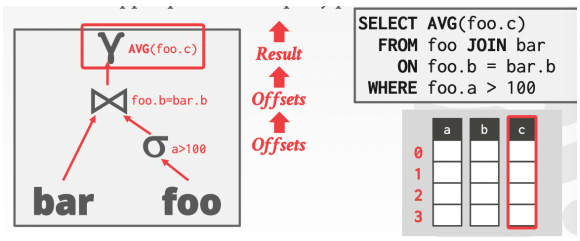
optimizations:

- prefetching
- buffer pool bypass
- parallelization
- heap clustering
- zone maps
- late meterialization

zone maps: pre-computed aggregates for the attributes values in a page. DBMS checks the zone map first to decide whether it wants to access the page



late meterialization: DSM DBMSs can delay stitching together tuples until the upper parts of the query plan



10.2.2 Index scan

The DBMS picks an index to find the tuples that the query needs
which index to use depends on

- what attributes the index contains
- what attributes the query references
- the attribute's value domains
- predicate composition
- whether the index has unique or non-unique keys

suppose that we have a single table with 100 tuples and two indexes: `age`, `dept`

```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```

scenario 1: there are 99 people under the age of 30 but only 2 people in the CS department
scenario 2: there are 99 people in the CS department but only 2 people under the age of 30
if there are multiple indexes that the DBMS can use for a query:

- compute sets of Record IDs using each matching index
- Combine these sets based on the query's predicates (union vs. intersect)
- retrieve the records and apply any remaining predicates

Postgres calls this **Bitmap Scan**
With an index on `age` and an index on `dept`

- we can retrieve the Record IDs satisfying `age<30` using the first
- then retrieve the Record IDs satisfying `dept='CS'` using the second
- take their intersection
- retrieve records and check `country='US'`

set intersection can be done with bitmaps, hash tables, or Bloom filters

10.3 Modification Queries

Operators that modify the database (`INSERT`, `UPDATE`, `DELETE`) are responsible for checking the constraints and updating indexes

`UPDATE/DELETE`:

- child operators pass Record IDs for the target tuples
- must keep track of previously seen tuples

`INSERT`:

- choice 1: materialize tuples inside of the operator
- choice 2: operator inserts any tuple passed in from child operators

Halloween Problem: anomaly where an update operation changes the physical location of a tuple, which causes a scan operator to visit the tuple multiple times

10.4 Expression Evaluation

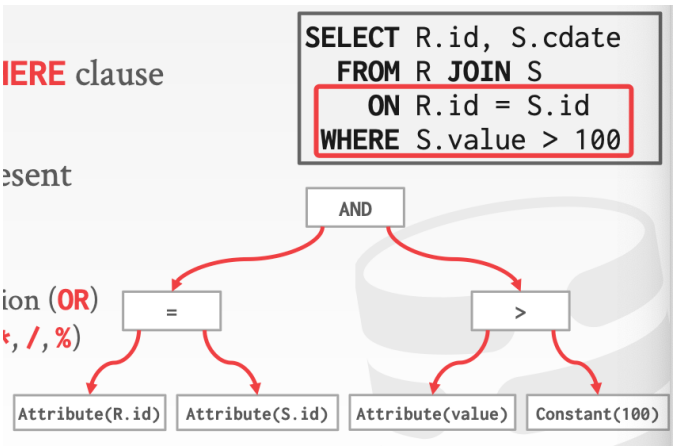
The DBMS represents a `WHERE` clause as an **expression tree**

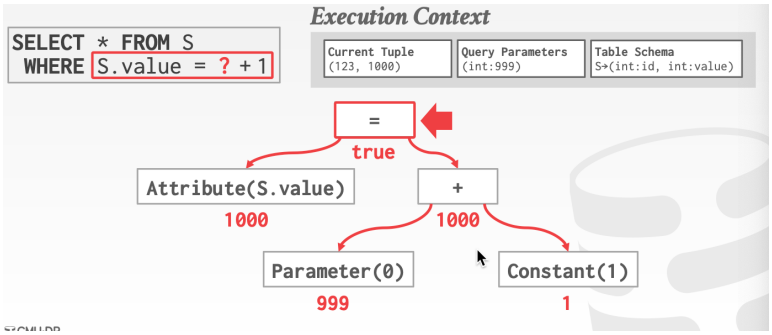
```
SELECT R.id, S.cdata
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

The nodes in the tree represent different expression types:

- comparisons (`=`, `<`, `>`, `!=`)

- conjunctions `AND`, disjunction `OR`
- arithmetic operators (`+`, `-`, `*`, `/`, `%`)
- constant values
- tuple attribute references





11 Query Execution 2

Parallel DBMSs

- resources are physically close to each other
- resources communicate over high-speed interconnect

- communication is assumed to be cheap and reliable

Distributed DBMSs

- resources can be far from each other
- resources communicate using slow interconnect
- communication cost and problems cannot be ignored

11.1 Process Models

A DBMS's **process model** defines how the system is architected to support concurrent requests from a multi-user application

A **worker** is the DBMS component that is responsible for executing tasks on behalf of the client and returning the results

1. Process per DBMS Worker

   each worker is a separate OS process

   - relies on OS scheduler
   - use shared-memory for global data structures
   - a process crash doesn't take down entire system
   - examples: IBM DB2, Postgres, oracle

2. Process Pool

   a worker uses any free process from the pool

   - still relies on OS scheduler and shared memory
   - bad for cpu cache locality
   - examples: IBM DB2, Postgres(2015)

3. Thread per DBMS Worker

   single process with multiple worker threads

   - DBMS manages its own scheduling
   - may or may not use a dispatcher thread
   - thread crash (may) kill the entire system
   - examples: IBM DB2, MSSQL, MySQL, Oracle(2014)

Advantages of a multi-threaded architecture

- less overhead per context switch
- do not have to manage shared memory

The thread per worker model does **not** mean that the DBMS supports intra-query parallelism

For each query plan, the DBMS decides where, when, and how to execute it

- how many tasks should i use
- how many CPU cores should it use
- what CPU core should the tasks execute on
- where should a task store its output

The DBMS **always** knows more than the OS
Inter-query: different queries are executed concurrently

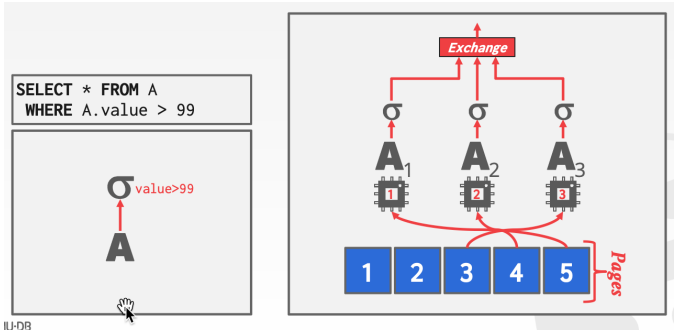
- increases throughput and reduces latency
- if queries are read-only, then this requires little coordination between queries
- if multiple queries are updating the database at the same time, then this is hard to do correctly

Intra-query: execute the operations of a single query in parallel

- decreases latency for long-running queries
- think of organization of operators in terms of **producer/consumer** paradigm
- there are parallel versions of every operator: can either have multiple threads access centralized data structures or use partitioning to divide work up

e.g., for parallel grace hash join, use a separate worker to perform the join for each level of buckets for R and S after partitioning

intra-query parallelism:



11.1.1 intra-operator (horizontal)

decompose operators into independent **fragments** that perform the same function on different subsets of data

the DBMS inserts an **exchange** operator into the query plan to coalesce/split results from multiple children/parent operators

exchange operator

1. exchange type 1 - **gather**: combine the results from multiple workers into a single output stream
2. exchange type 2 - **distribute**: split a single stream into multiple output streams
3. exchange type 3 - **repartition**: shuffle multiple input streams across multiple output streams

11.1.2 inter-operator (vertical)

operations are overlapped in order to pipeline data from one stage to the next without materialization

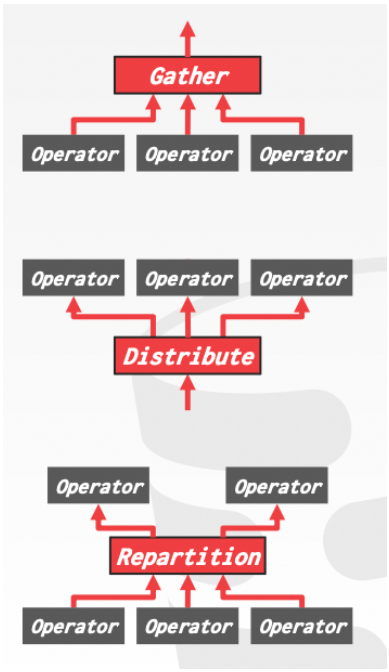
workers execute operators from different segments of a query plan at the same time

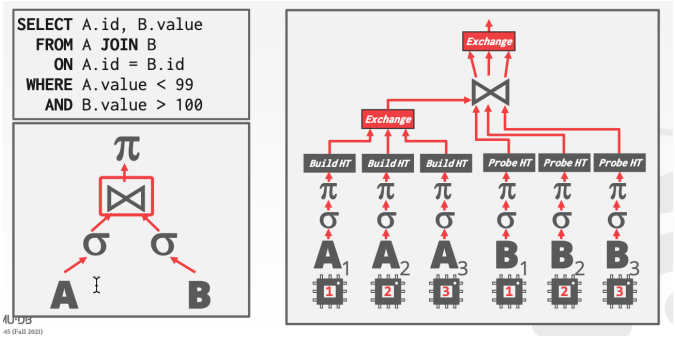
also called **pipeline parallelism**

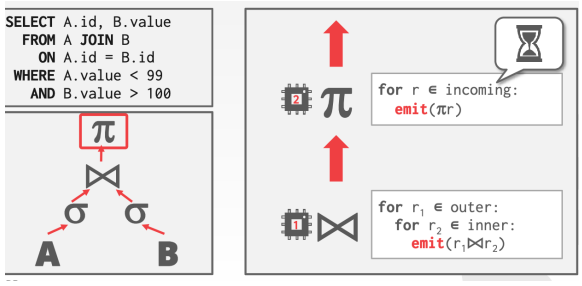
11.1.3 bushy

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

still need exchange operators to combine intermediate results from segments

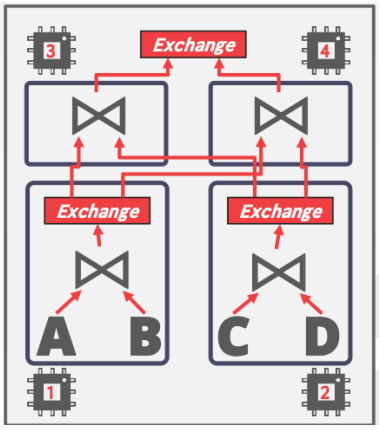

Gather
Operator
Operator
Operator
Operator
Operator
Operator
Distribute
Operator
Operator
Repartition
Operator
Operator
Operator


SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
π
σ
σ
A
B
Exchange
Exchange
Build HT
Build HT
Build HT
Probe HT
Probe HT
Probe HT
A1
A2
A3
B1
B2
B3


SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
π
σ
σ
A
B
for r ∈ incoming:
emit(πr)
for r1 ∈ outer:
for r2 ∈ inner:
emit(r1⋈r2)

for

```
SELECT *
FROM A JOIN B JOIN C JOIN D
```



11.2 Execution Parallelism

11.2.1 I/O parallelism

split the DBMS across multiple storage devices

- multiple disks per database
- one database per disk
- one relation per disk
- split relation across multiple disks

partitioning: split single logical table into disjoint physical segments that are stored/managed separately
partitioning should be transparent to the application

- the application should only access logical tables and not have to worry about how things are physically stored

vertical partitioning: store a table's attributes in a separate location

11.3 I/O Parallelism

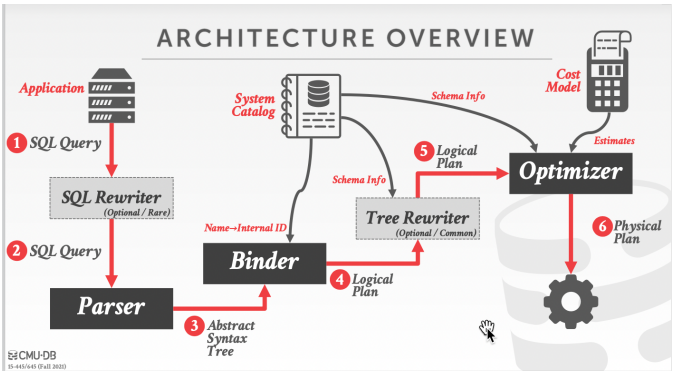
12 Optimization 1

Heuristics/Rules

- rewrite the query to remove stupid/inefficient things
- these techniques may need to examine catalog, but they do **not** need to examine data

Cost-based search

- use a model to estimate the cost of executing a plan
- evaluate multiple equivalence plans for a query and pick the one with the lowest cost



the optimizer generates a mapping of a logical algebra expression to the optimal equivalent physical algebra expression

physical operators define a specific execution strategy using an access path

- they can depend on the physical format of the data that they process
- not always a 1:1 mapping from logical to physical

query optimization is NP-Hard

12.1 Relational Algebra Equivalences

Two relational algebra expressions are **equivalent** if they generate the same set of tuples

This is often called **query rewriting**

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

$\pi_{\text{name,cid}}(\sigma_{\text{grade='A'}}(\text{student} \bowtie \text{enrolled}))$, which is equivalent to $\pi_{\text{name,cid}}(\text{student} \bowtie (\sigma_{\text{grade='A'}}(\text{enrolled})))$,

Selections:

- perform filters as early as possible
- break a complex predicate, and push down

$$\sigma_{p_1 \wedge \cdots \wedge p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots \sigma_{p_n}(R)))$$

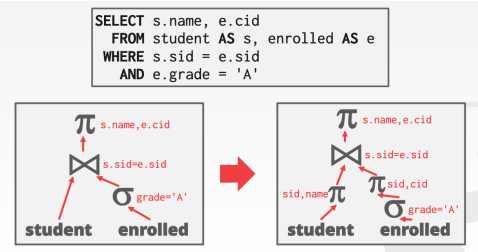
Joins:

- commutative, associative

The number of different join orderings for an n-way join is a **Catalan Number**

Projections:

- perform them early to create smaller tuples and reduce intermediate results
- project out all attributes except the ones requested or required



12.2 Logical Query Optimization

Transform a logical plan into an equivalent logical plan using pattern matching rules

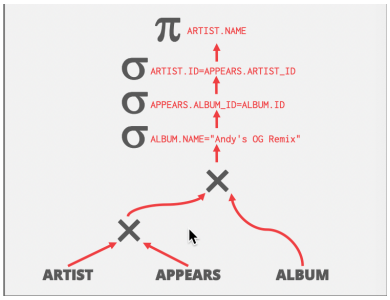
The goal is to increase the likelihood of enumerating the optimal plan in the search

12.2.1 Split Conjunctive Predicates

Consider

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID = APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID = ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```

Decompose predicates into their simplest forms to make it easier for the optimizer to move them around



12.2.2 Predicate Pushdown

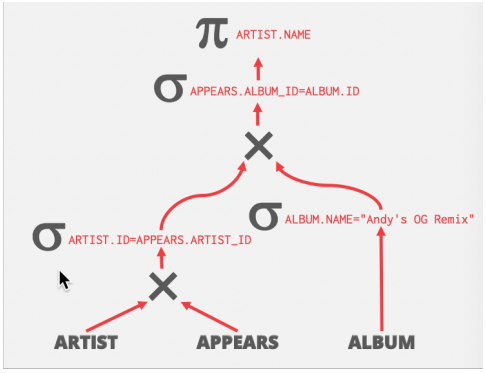
Move the predicate to the lowest applicable point in the plan

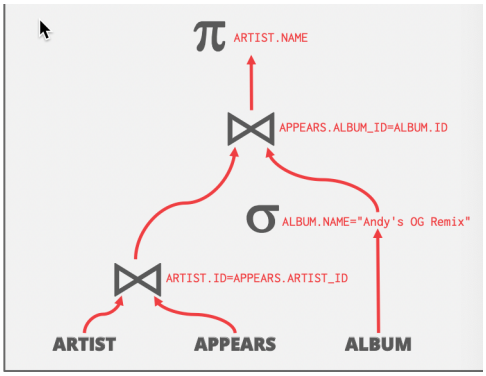
12.2.3 Replace Cartesian Products with Joins

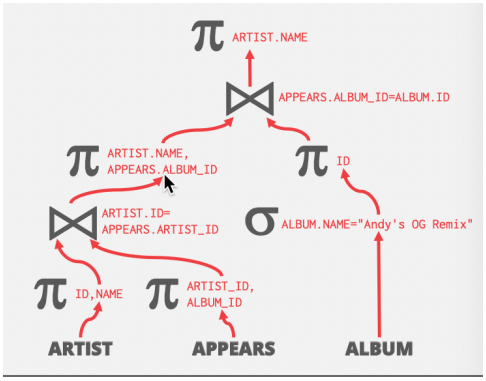
Replace all Cartesian Products with inner joins using the join predicates

12.2.4 Projection Pushdown

Eliminate redundant attributes before pipeline breakers to reduce materialization cost


π ARTIST.NAME
σ APPEARS.ALBUM_ID=ALBUM.ID
×
σ ALBUM.NAME="Andy's OG Remix"
σ ARTIST.ID=APPEARS.ARTIST_ID
×
ARTIST
APPEARS
ALBUM


π ARTIST.NAME
⋈ APPEARS.ALBUM_ID=ALBUM.ID
σ ALBUM.NAME="Andy's OG Remix"
⋈ ARTIST.ID=APPEARS.ARTIST_ID
ARTIST
APPEARS
ALBUM


π ARTIST.NAME
⋈ APPEARS.ALBUM_ID=ALBUM.ID
π ARTIST.NAME, APPEARS.ALBUM_ID
π ID
⋈ ARTIST.ID= APPEARS.ARTIST_ID
σ ALBUM.NAME="Andy's OG Remix"
π ID,NAME
π ARTIST_ID, ALBUM_ID
ARTIST
APPEARS
ALBUM

12.3 Nested 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

- rewrite to de-correlate and/or flattern them
- decompose nested query and store result to temporary table

12.3.1 Rewrite

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

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

12.3.2 Decompose

```
SELECT S.sid, MIN(R.day)
FROM sailors S, reserves R, boats B
WHERE S.sid = R.sid
AND R.bid = B.bid
AND B.color = 'red'
AND S.rating = (SELECT MAX(S2.rating)
GROUP BY S.sid HAVING COUNT(*) > 1
```

For each sailor with the highest rating (over all sailors) and at least two reservations for red boats, find the sailor id and the earliest data on which the sailor has a reservation for a red boat

For harder queries, the optimizer breaks up queries into blocks and then concentrate on one block at a time. Sub-queries are written to a temporary table that are discarded after the query finishes

12.4 Expression Rewriting

An optimizer transforms a query's expressions (e.g., `WHERE` clause predicates) into the optimal/minimal set of expressions

Implemented using if/then/else clauses or a pattern-matching rule engine

- search for expressions that match a pattern
- when a match is found, rewrite the expression
- halt if there are no more rules that match

Impossible/unnecessary predicates
join elimination

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

join elimination with sub-query

```
SELECT * FROM A AS A1
 WHERE EXISTS (SELECT val FROM A AS A2
                 WHERE A1.id = A2.id);
```

merging predicates

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 100
    OR val BETWEEN 50 AND 150;
```

12.5 Cost Model

1. Physical costs
    - predict CPU cycles, I/O, cache misses, RAM consumption, prefetching, etc
    - depends heavily on hardware
2. logical costs
    - estimate result sizes per operator
    - independent of the operator algorithm

- need estimations for operator result sizes

3. algorithmetic costs

disk-based DBMS cost model: the number of disk accesses will always dominate the execution time of a query

- CPU costs are negligible
- must consider sequential vs. random I/O

The is easier to model if the DBMS has full control over buffer management

postgres cost model: use a combination of CPU and I/O costs that are weighted by "magic" constant factors

default settings are for a disk-resident database without a lot of memory

- processing a tuple in memory is 400x faster than reading a tuple from disk
- sequential I/O is 4x faster than random I/O

IBM DB2 cost model:

- database characteristics in system catalogs
- hardware environment
- storage device characteristics
- communications bandwidth
- memory resources
- concurrency environment

12.6 More cost estimation

The DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog

different systems update them at different times

manual invocations:

- postgres/sqlite: `ANALYZE`
- oracle/mysql: `ANALYZE TABLE`

- SQL server: `UPDATE STATISTICS`
- DB2: `RUNSTATS`

For each relation R, the DBMS maintains the following information

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

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

Equality predicates on unique keys are easy to estimate

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

computing the logical cost of complex predicates is more difficult

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

The **selectivity** (`sel`) of a predicate P is the fraction of tuples that qualify
Formula depends on type of predicate:

- equality
- range
- negation
- conjunction
- disjunction

Assume that `V(age,people)` has five distinct values (0-4) and $N_R = 5$
Equality Predicate: `A=constant`

- $sel(A = constant) = SC(P)/N_R$

Range Predicate:

- $sel(A \geq a) = (A_{max} - a + 1)/(A_{max} - A_{min} + 1)$

Negation Query:

- $sel(\neg P) = 1 - sel(P)$

Conjunction:

- $sel(P_1 \wedge P_2) = sel(P_1) \cdot sel(P_2)$
- assumes predicates are **independent**

Disjunction:

- $sel(P_1 \vee P_2) = sel(P_1) + sel(P_2) - sel(P_1) \cdot sel(P_2)$
- also assumes independence

Given a join of R and S, what is the range of possible result sizes in # of tuples. In other words, for a given tuple of R, how many tuples of S will it match?

Assume each key in the inner relation will exist in the outer table.

General case: $R_{cols} \cap S_{cols} = \{A\}$, where A is not a primary key for either table

- match each R-tuple with S-tuples:

$$estSize \approx N_r \cdot N_s / V(A, S)$$

- for S

$$estSize \approx N_r \cdot N_s / V(A, R)$$

Overall, $estSize \approx N_R \cdot N_s / \max(\{V(A, S), V(A, R)\})$

For select cardinality, we have three assumptions

1. uniform data
2. independent predicates
3. inclusion principle

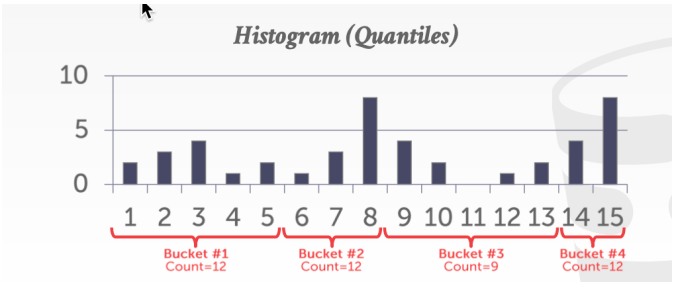
   the domain of join keys overlap s.t. each key in the inner relation will also exist in the outer table

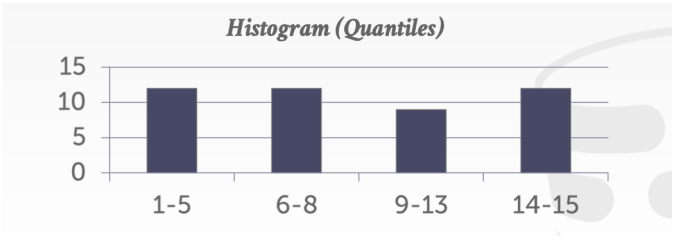
For correlated attributes, consider a database of automobiles, # of Makes=10, # of Models = 100 and the following query `make="Honda" AND model="Accord"`

With the independent and uniformity assumptions, the selectivity is

$$1/10 \times 1/100 = 0.001$$

But since only Honda makes Accords the real selectivity is 0.01





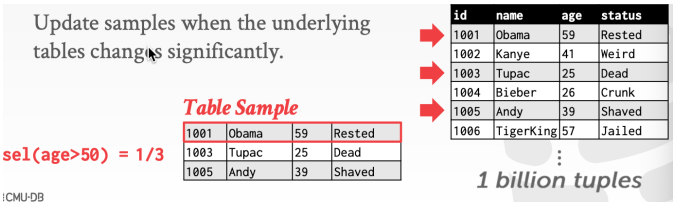
For non-uniform data, we may use equi-width histogram, and we can vary the width of buckets so that the total number of occurrences for each bucket is roughly the same

Sketches is a probabilistic data structures that generate approximate statistics about a data set. And Cost-model can replace histogram with sketches to improve its selectivity estimate accuracy

Most common examples:

- Count-Min Sketch
- HyperLogLog

Modern DBMSs also collect samples from tables to estimate selectivities. Update samples when the underlying tables changes significantly



12.7 plan enumeration

Now that we can (roughly) estimate the selectivity of predicates, and subsequently the cost of query plans, what can we do with them?

After performing rule-based rewriting, the DBMS will enumerate different plans for the query and estimate their costs.

- single relation
- multiple relations
- nested sub-queries

12.7.1 single relation

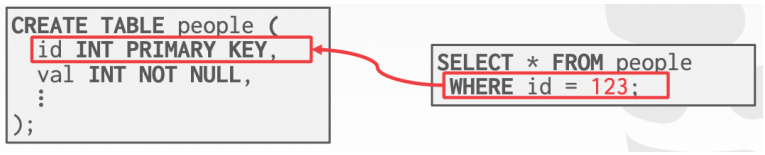
For single-relation query planning, pick the best access method

- sequential scan
- binary search
- index scan

Simple heuristics are often good enough for this. OLTP queries are especially easy

Query planning for OLTP queries is easy because they are **sargable** (Search Argument Able)

- it is usually just picking the best index
- joins are almost always on foreign key relationships with a small cardinality
- can be implemented with simple heuristics



12.7.2 multi-relation

fundamental decision in System R: only consider left-deep join trees

use **dynamic programming** to reduce the number of cost estimations

how to generate plans for search algorithm:

- enumerate relation orderings
- enumerate join algorithm choices
- enumerate access method choices

No real DBMSs does it this way. It's actually more messy

Postgres optimizer examines all types of joins trees, and has two optimizer implementations

1. traditional dynamic programming approach
2. genetic query optimizer

Postgres uses the traditional algorithm when # of tables in query is **less** than 12 and switches to GEQO when there are 12 or more

13 Concurrency Control

Motivation:

We both change the same record in a table at the same time. **How to avoid race condition?** (Concurrency Control)

You transfer $100 between bank accounts but there is a power failure. **What is the correct database state** (Recovery)

They are based on the concept of transactions with **ACID** properties

A **transaction** is the execution of a sequence of one or more operations on a database to perform some higher-level function

It is the basic unit of changes in a DBMS:

- partial transactions are not allowed

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

Transaction:

- check whether Lin has $100
- Deduct $100 from his account

- Add $100 to his promotor account

Strawman system:
Execute each txn one-by-one as they arrive at the DBMS

- One and only one txn can be running at the same time in the DBMS

Before a txn starts, copy the entire database to a new file and make all changes to that file

- if the txn completes successfully, overwrite the original file with the new one
- if the txn fails, just remove the dirty copy

Problem Statements: how to allow concurrent execution of independent transactions with better utilization/throughput and increasing response times to users

We need formal correctness criteria to determine whether an interleaving is valid

A **Database** is a fixed set of named data objects, $A, B, C, ...$

A **transaction** is a sequence of read and write operation $(R(A))$, $W(B)$

A new txn starts with the `BEGIN` command and stops with either `COMMIT` or `ABORT`:

- if commit, the DBMS either saves all the txn's changes or aborts it
- if abort, all changes are undone so that it's like as if the txn never executed at all

Abort can be either self-inflicted or caused by the DBMS

Correctness Criteria: **ACID**

- **Atomicity**: all actions in the txn happen, or none happen

  "all or nothing"

- **Consistency**: if each txn is consistent and the DB starts consistency, then it ends up consistency

  "it looks correct to me"

- **Isolation**: execution of one txn is isolated from that of other txns

  "as if alone"

- **Durability**: if a txn commits, its effects persist

  "survive failures"

13.1 Atomicity

Two possible outcomes of executing a txn

- commit after completing all its actions
- abort (or be aborted by the DBMS) after executing some actions

DBMS guarantees that txns are **atomic**

- from user's point of view: txn always either executes all its actions or executes no actions at all

Scenario 1: we take $100 out of Lin's account but then the DBMS aborts the txn before we transfer it

Scenario 2: We take $100 out of Lin's account but then there is a power failure before we transfer it

Approach 1: **logging**

- DBMS logs all actions so that it can undo the actions of aborted transactions
- maintain undo records both in memory and on disk

Logging is used by almost every DBMS

Approach 2: **Shadow Paging**

- DBMS makes copies of pages and txns make changes to those copies. Only when the txn commits it the page made visible to others
- Originally from System R

Few systems do this: CouchDB, LMDB

13.2 Consistency

database consistency, transaction consistency

13.3 Isolation

isolation of transactions, users submit txns, and each txn executes as if it was running by itself

But the DBMS achieves concurrency by interleaving the actions (read-/writes of DB objects) of txns

We need a way to interleave txns but still make it appear as if they ran one-at-a-time

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

Two categories of protocols:

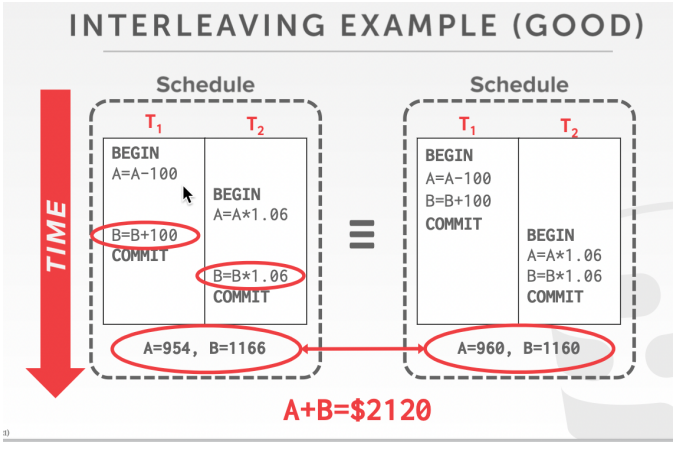
- **pessimistic**: don't let problems arise in the first place
- **optimistic**: assume conflicts are rare, deal with them after they happen

Assume at first A and B each have $1000

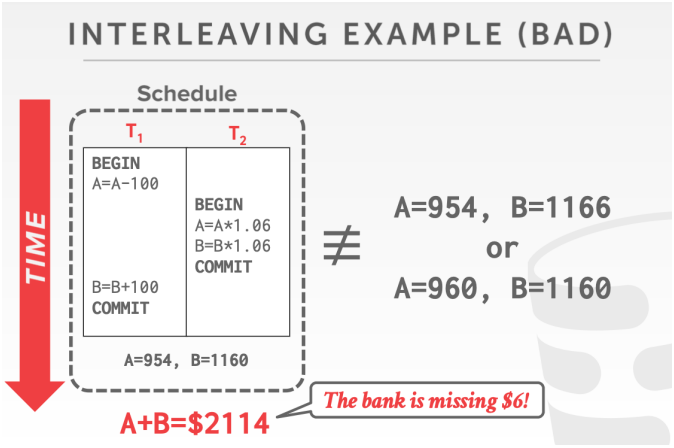
```
/* T1 */
BEGIN
A = A + 100
B = B - 100
COMMIT

/* T2 */
BEGIN
A = A * 1.06
B = B * 1.06
COMMIT
```

The outcome depends on whether `T1` executes before `T2` or vice versa

We interleave txns to maximize concurrency



How do we judge whether a schedule is correct?



If the schedule 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 executing the first schedule is identicial to the effect of executing the second schedule

If each transaction preserves consistency, every serializable schedule preserves consistency

Two operations **conflict** if

1. they are by different transactions
2. they are on the same object and at least one of them is a write

So we have read-write conflicts, write-read conflicts and write-write conflicts

Given these conflicts, we now can understand what it means for a schedule to be serializable

- this is to check whether schedules are correct

The are different levels of serializability

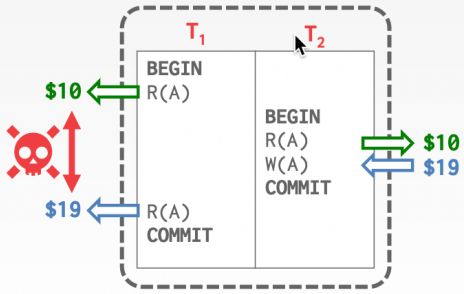
- conflict serializability (Most DBMSs try to support this)
- view serializability (No DBMS can do this)

Two schedules are **conflict equivalent** iff

- they involve the same actions of the same transactions

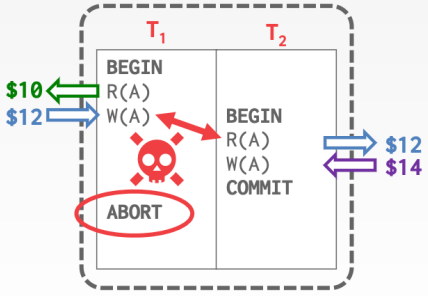
READ-WRITE CONFLICTS

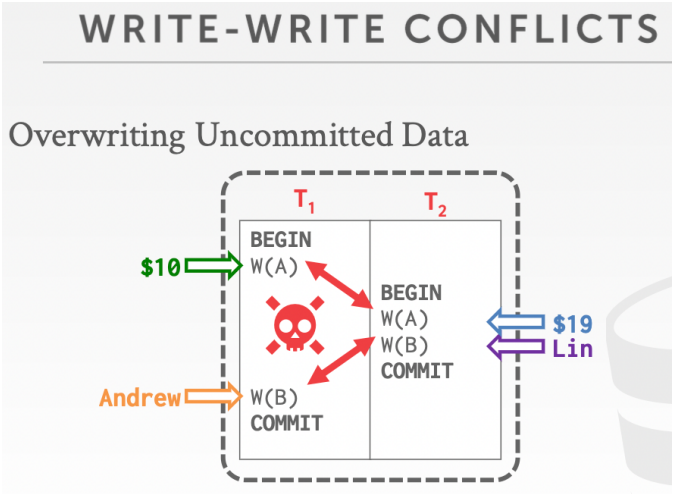
Unrepeatable Reads



WRITE-READ CONFLICTS

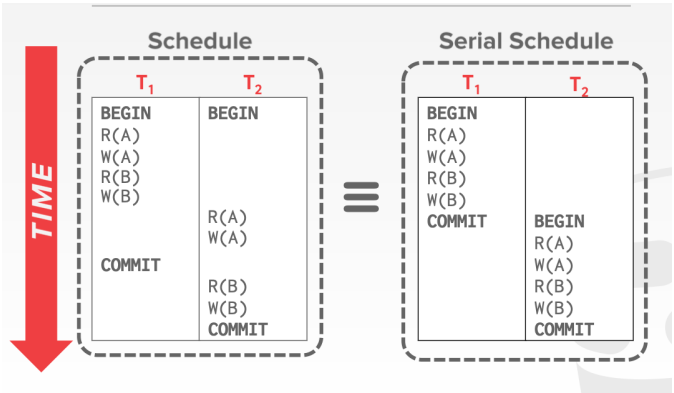
Reading Uncommitted Data ("Dirty Reads")





- every pair of conflicting actions is ordered the same way

Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule, that is, if you can transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions

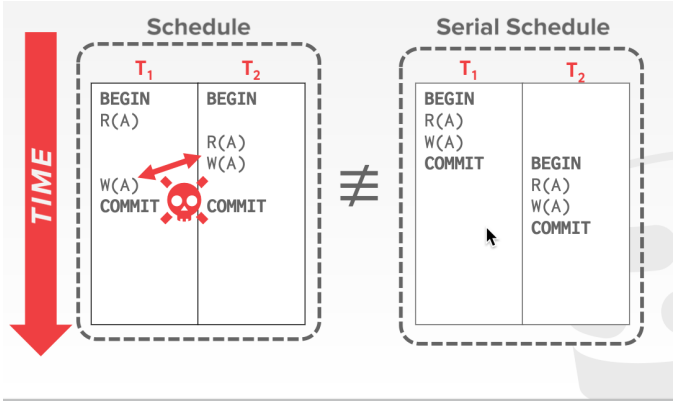


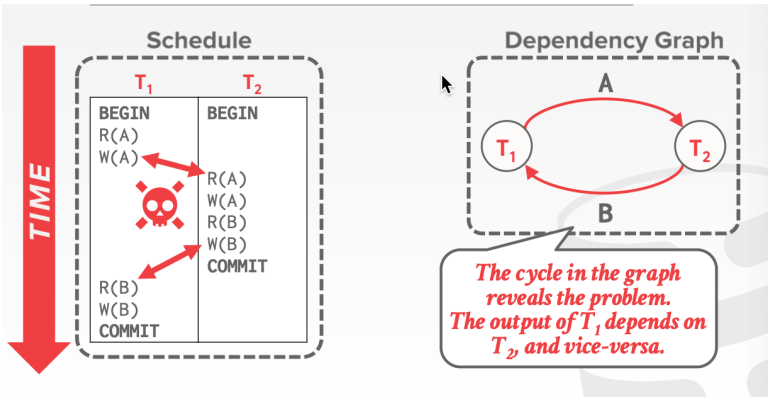
Swapping operations is easy when there are only two txns in the schedule. It's cumbersome when there are many txns? **Dependency Graphs**

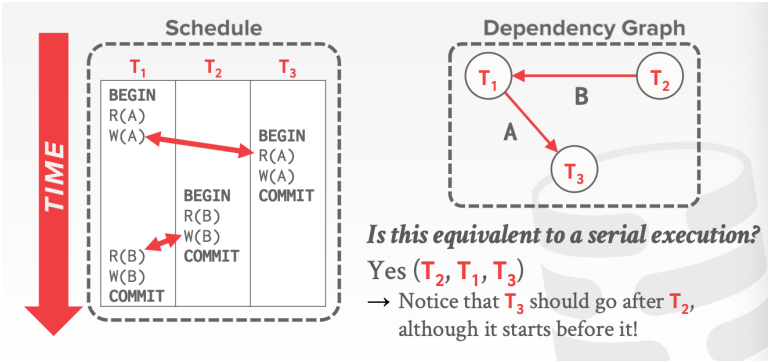
One node per txn. Edge from T_i to $T_j$if

- an operation O_i of T_i conflicts with an operation O_j of T_J
- O_i appears earlier in the schedule than O_j

Also known as a **prescedence graph**

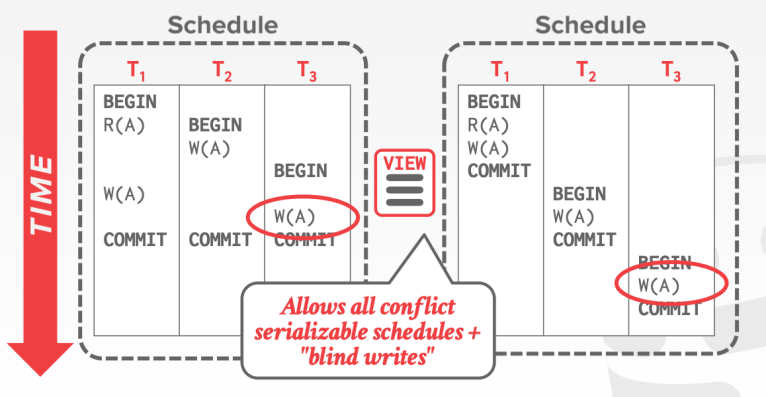

Schedule
Serial Schedule
TIME
T1
T2
BEGIN
R(A)
W(A)
COMMIT
BEGIN
R(A)
W(A)
COMMIT
≠
BEGIN
R(A)
W(A)
COMMIT
BEGIN
R(A)
W(A)
COMMIT


Schedule
Dependency Graph
TIME
T1
T2
BEGIN
R(A)
W(A)
R(A)
W(A)
R(B)
W(B)
COMMIT
R(B)
W(B)
COMMIT
A
B
The cycle in the graph reveals the problem.
The output of T1 depends on T2, and vice-versa.


Schedule
Dependency Graph
TIME
T1
T2
T3
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
BEGIN
R(A)
W(A)
COMMIT
R(B)
W(B)
COMMIT
B
A
Is this equivalent to a serial execution?
Yes (T2, T1, T3)
→ Notice that T3 should go after T2, although it starts before it!

Schedules S_1 and S_2 are view equivalent if

- if T_1 reads initial value of A in S_1, then T_1 also reads initial value of A in S_2
- if T_1 reads value of A written by T_2 in S_1, then T_1 also reads value of A written by T_2 in S_2
- If T_1 writes final value of A in S_1, then T_1 also writes final value of A in S_2



View Serializability allows for more schedules than **Conflict Serializability** does, but is difficult to enforce efficiently

In practice, **Conflict Serializability** is what systems support because it can be enforced efficiently

Serial $\subseteq$ Conflict Serializable $\subseteq$ View Serializable

13.4 Durability

All the changes of committed transactions should be persistent

14 Two-Phase Locking

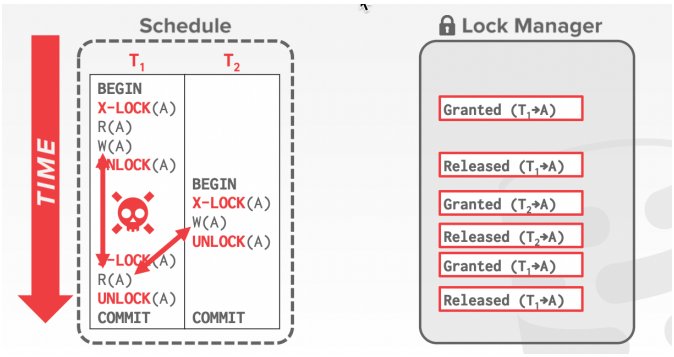
14.1 Lock Types

S-LOCK: shared locks for reads

X-LOCK: exclusive locks for writes

	shared	exclusive
shared	✓	×
exclusive	×	×

14.2 Two-Phase Locking



Two-phase locking (2PL) is a concurrency control protocol that determines whether a txn can access an object in the database on the fly

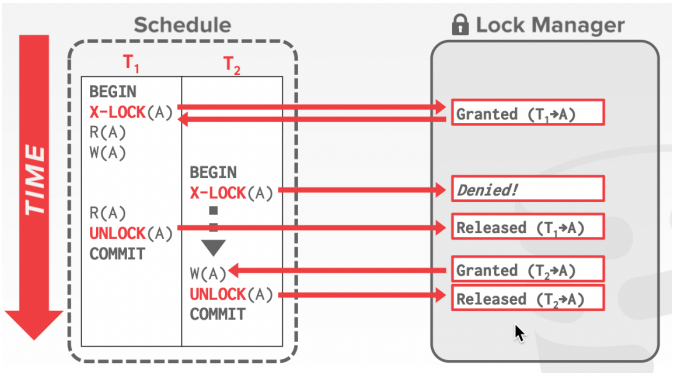
The protocol does **not** need to know all the queries that a txn will execute ahead of time

Phase 1: Growing

- each txn requests the locks that it needs from the DBMS's lock manager
- the lock manager grants/denies lock requests

Phase 2: Shrinking

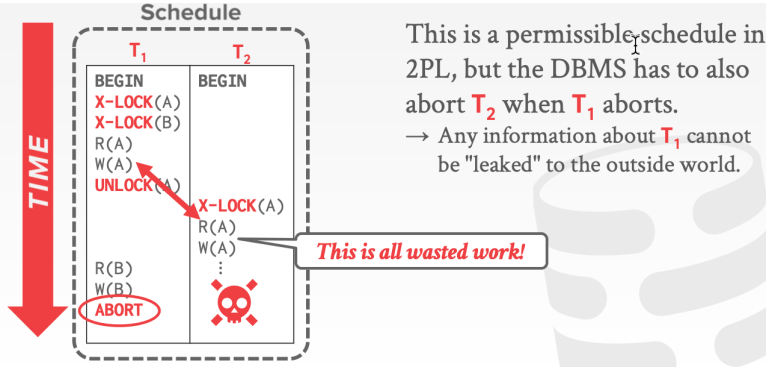
- the txn is allowed to only release locks that it previously acquired. It cannot acquire new locks



2PL on its own is sufficient to guarantee conflict serializability

- it generates schedules whose precedence graph is acylic

  But it is subject to **cascading aborts**



  There are potential schedules that are serializable but would not be allowed by 2PL

  – locking limits concurrency

may still have "dirty reads"

- solution: **Strong Strict 2PL** (aka **Rigorous 2PL**)

May lead to deadlock

- solution: **Detection** or **Prevention**

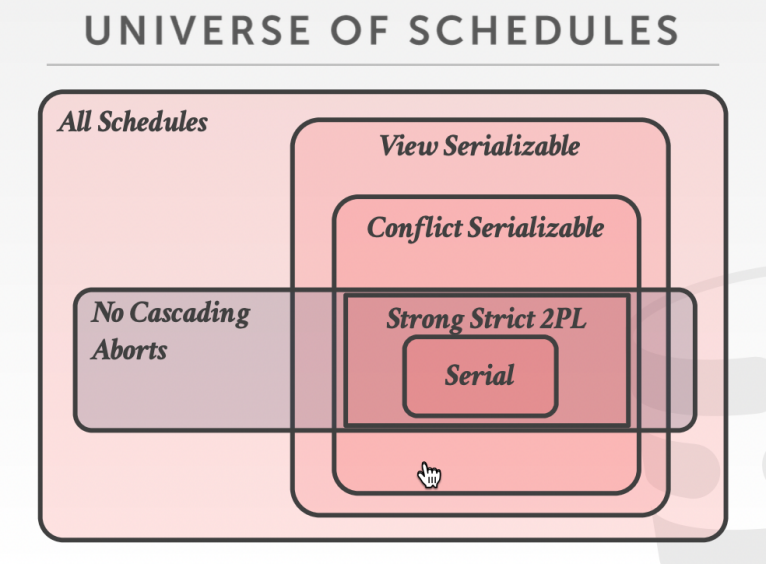
Strong strict two-phase locking: the txn is only allowed to release locks after it has ended, i.e., committed or aborted

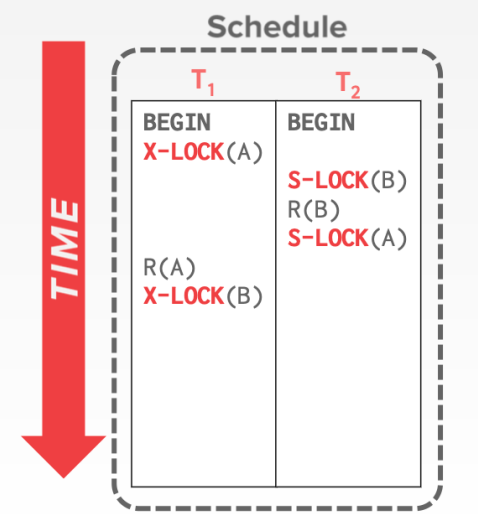
Allows only conflict serializable schedules, but it is often stronger than needed for some apps

A schedule is **strict** if a value written by a txn is not read or overwritten by other txns until that txn finishes

Advantages:

- does not incur cascading aborts
- aborted txns can be undone by just restoring original values of modified tuples


UNIVERSE OF SCHEDULES
All Schedules
View Serializable
Conflict Serializable
No Cascading Aborts
Strong Strict 2PL
Serial


Schedule
TIME
T1
T2
BEGIN
X-LOCK(A)
R(A)
X-LOCK(B)
BEGIN
S-LOCK(B)
R(B)
S-LOCK(A)

14.3 Deadlock Detection + Prevention

A **deadlock** is a cycle of transactions waiting for locks to be released by each other

Two ways of dealing with deadlocks

1. deadlock detection
2. deadlock prevention

14.3.1 Deadlock detection

The DBMS creates a **wait-for** graph to keep track of what locks each txn is waiting to acquire:

- nodes are transactions
- each from T_i to T_j if T_i is waiting for T_j to release a lock

When the DBMS detects a deadlock, it will select a "victim" txn to rollback to break the cycle

The victim txn will either restart or abort depending on how it was invoked

There is a trade-off between the frequency of checking for deadlocks and how long txns have to wait before deadlocks are broken

Selecting the proper victim depends on a lot of different variables

- by age
- by progress
- by the # of items already locked
- by the # of txns that we have to rollback with it

After select a victim txn to abort, the DBMS can also decide on how far to rollback the txn's changes - completely/minimally

14.3.2 Deadlock prevention

When a txn tried to acquire a lock that is held by another txn, the DBMS kills one of them to prevent a deadlock

This approach does *not* require a **waits-for** graph or detection algorithm

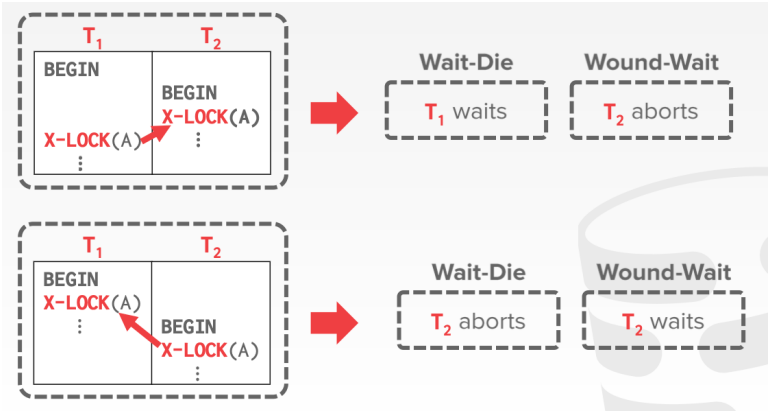
Assign priorities based on timestamps: old timestamp = higher priority

Wait-Die (Old Waits for Young)

- if *requesting txn* has higher priority than *holding txn*, then *requesting txn* waits for *holding txn*
- otherwise *requesting txn* aborts

Wound-Wait (Yound waits for old)

- if *requesting txn* has higher priority than *holding txn*, then *holding txn* aborts and releases lock
- otherwise *requesting txn* waits



Why do these schemes guarantee no deadlocks?
only one "type" of direction allowed when waiting for a lock
When a txn restarts, what is its priority?
Its original timestamp(so you have enough priority)

14.4 Hierarchical Locking

All these examples have a one-to-one mapping from database objects to locks.

If a txn wants to update one billion tuples, then it must acquire one billion locks

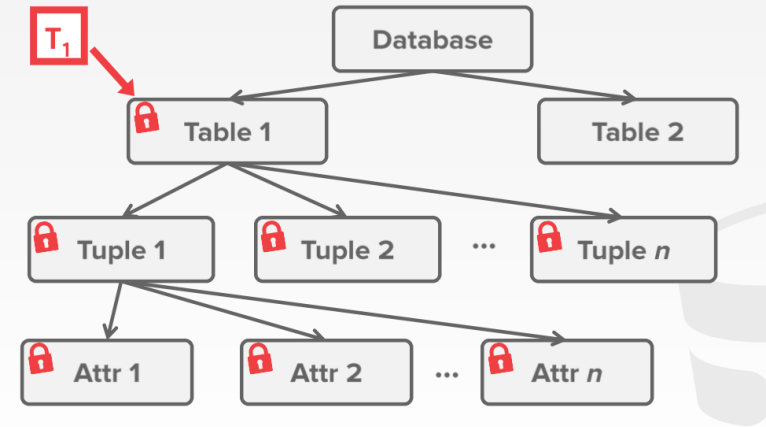
Acquiring locks is a more expensive operation than acquiring a latch even if that lock is available

When a txn wants to acquire a "lock", the DBMS can decide the granularity of that lock

The DBMS should ideally obtain fewest number of locks that a txn needs
Trade-off between parallelism versus overhead

- fewer locks, larger granularity vs. More Lockers, smaller granularity

Database lock hierarchy:



An **intention lock** allows a higher-level node to be locked in **shared** or **exclusive** mode without having to check all descendent nodes

If a node is locked in an intention mode, then some txn is doing explicit locking at a lower level in the tree

Intention locks:

- **intention-shared** (`IS`): indicates explicit locking at lower level with shared locks
- **intention-exclusive** (`IX`): indicates explicit locking at lower level with exclusive locks
- **shared+intention-exclusive** (`SIX`): the subtree rooted by that node is is locked explicitly in **shared** mode and explicit is being done at a lower level with **exclusive-mode** locks

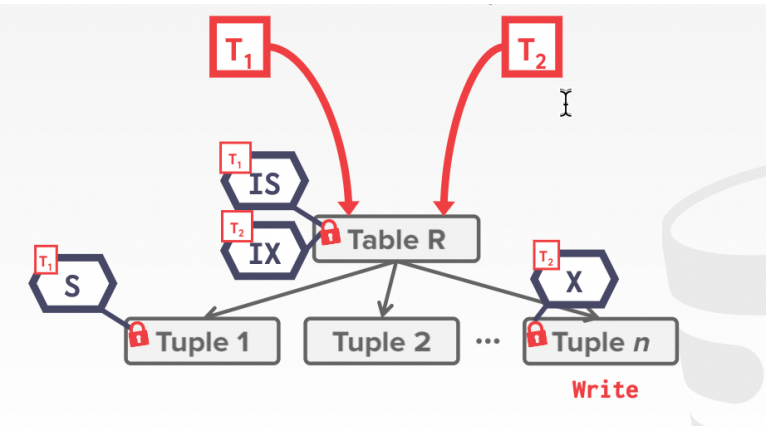
	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×		×	×
X	×	×	×	×	×

Locking protocol:

- each txn obtains appropriate lock at highest level of the database hierarchy
- to get `S` or `IS` lock on a node, the txn must hold at least IS on parent node
- to get `X`, `IX`, or `SIX` on a node, must hold at least IX on parent node

Example:
T_1: Get the balance of Lin's shady off-shore bank account
T_2: Increase Andrew's bank account balance by 1%

- exclusive + shared for leaf nodes of lock tree
- special intention locks for higher levels



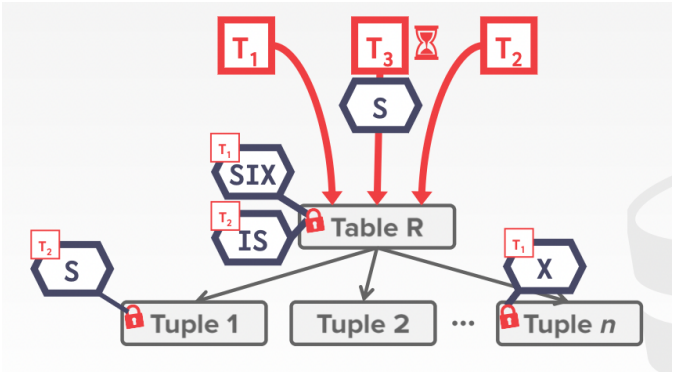
Assume three txns execute at same time:

- T_1: scan `R` and update few tuples
- T_2: read a single tuple in `R`
- T_3: Scan all tuples in `R`

  Hierarchical locks are useful in practice as each txn only needs a few locks

  Intention locks help improve concurrency:

  - intention-shared: intent to get S locks at finer granularity
  - intention-exclusive: intent to get X locks at finer granularity
  - shared+intention-exclusive: like S and IX at the same time



14.5 Conclusion

2PL is used in almost DBMS

Automatically generates correct interleaving:

- locks + protocol (2PL, SS2PL)
- deadlock detection + handling
- deadlock prevention

15 Timestamp Ordering Concurrency Control

Concurrency control approaches:

- Two-Phase locking (pessimistic)
- Timestamp Ordering (optimistic)

Use timestamps to determine the serializability order of txns

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

Each txn T_i is assigned a unique fixed timestamp that is monotonically increasing

- let $TS(T_i)$ be the timestamp allocated to txn T_i
- different schemes assign timestamps at different times during the txn

Multiple implementation strategies

- system lock

- logical counter
- hybrid

15.1 Basic Timestamp Ordering (T/O) protocol

Txns read and write objects without locks

Each object X is tagged with timestamp of the last txn that successfully did read/write:

- `W-TS(X)` write timestamp on X
- `R-TS(X)` read timestamp on X

Check timestamps for every operation: if txn tries to access an object "from the future", it aborts and restarts

Read:

If `TS(Ti)` $<$ `W-TS(X)`, this violates timestamp order of T_i with regard to the write of X, then abort T_i and restart it with a new TS

Else, allow T_i to read X, update `R-TS(X)` with `max(R-TS(X),TS(\(T_i\)))`

15.2 Optimistic Concurrency Control

15.3 Isolation Levels

16 Lab notes

16.1 project 3

We will use the iterator query processing model (i.e., the Volcano model). Recall that in this model, every query plan executor implements a Next function. When the DBMS invokes an executor's Next function, the executor returns either

1. a single tuple or
2. an indicator that there are

no more tuples. With this approach, each executor implements a loop that continues calling Next on its children to retrieve tuples and process them one-by-one.

In BusTub's implementation of the iterator model, the Next function for each executor returns a record identifier (RID) in addition to a tuple. A record identifier serves as a unique identifier for the tuple relative to the table to which it belongs.

17 Homework

17.1 3

1. sorting algorithms
   (a) 10
   (b) 120m
   (c) 2450
   (d) 15
   (e) 154472232
2. Join algorithms
   (a) 56400
   (b) 55400
   (c) 7200 3600
   (d) b
   (e) i. 5600
   ii. 8800
   iii. 3600
   iv. 3160000
   v. 3000

17.2 4

1. (a) no
   (b) yes
   (c) yes
   (d) no ?
   (e) yes
2. (a) no
   (b) B
   (c) no
   (d) T1 T2 T3
   (e) yes

3. deadlock detection and prevention
   (a) i. S(A) S(B) X(B)
       ii. C D
   (b) i. g g b g b b
       ii. a
       iii. d
   (c) i. g b b g b b
       ii. d
       iii. b
       iv. g a a g g -
       v. g b b g a -
4. (a) c
   (b) c b
   (c) a
   (d) d
   (e) c a