Database and Optimizing Storage

Mainack Mondal
Sandip Chakraborty

CS60203
Autumn 2024

Today's class

- **Why care about a database?**

- Components of a database
  - Compute (query) and Storage

- Database Internals: Overview
  - OLAP and OLTP databases
  - Storage models
  - In-database optimizations

- Performance Pitfalls
  - Doing unnecessary work- ORMs, Absent or bad indexes, Layout
  - Read/Write amplification

Why care about a database?

Why use a database? And not the file system?

“I want to store data to the disk”
- use a file system

“I want a performant way to store data to the disk”
- use a better file system *(better may vary with use-cases)*

“But I want to distribute this data across nodes!”
- use a distributed file system

So, why not *just* use files?

The case for the database: Need of Guarantees

Data has meaning, so you need to uphold it

- Imagine that you are building a payments app
  - you store every transaction in some *storage system*
- So a transaction in storage system has a meaning for your application logic
  - (eg, Ajay paid Rahul 100 bucks), just like assigning to a variable

The case for the database: Need of Guarantees

Data has meaning, so you need to uphold it

- Imagine that you are building a payments app
  - you store every transaction in some *storage system*
- So a transaction in storage system has a meaning for your application logic
  - (eg, Ajay paid Rahul 100 bucks), just like assigning to a variable

this system must satisfy a set of guarantees, to ensure correctness!
Atomicity, **Consistency**, **Isolation**, and **Durability (ACID)**

Do file system provide **guarantees you need?**

If so, go right ahead and use plain old files! Else use a database that does

Note, this is not a rhetoric, many data systems (eg, Kafka, Hadoop), use files extensively for most/all work

The case for the database: Using the data

Data has meaning: how you use data can help you optimize

- Imagine that you are building a payments app, you store every transaction in some *storage system*
- Need: average of amounts of all the transfers from India to Canada

What do you think is the best way to store this data on the disk?

Do you think it'll be good to store all the **amounts together** or maybe all the data of a **transaction together**?

The operations you perform on stored data should determine how it is stored!

Today's class

- Why care about a database?

- **Components of a database**
  - Compute (query) and Storage

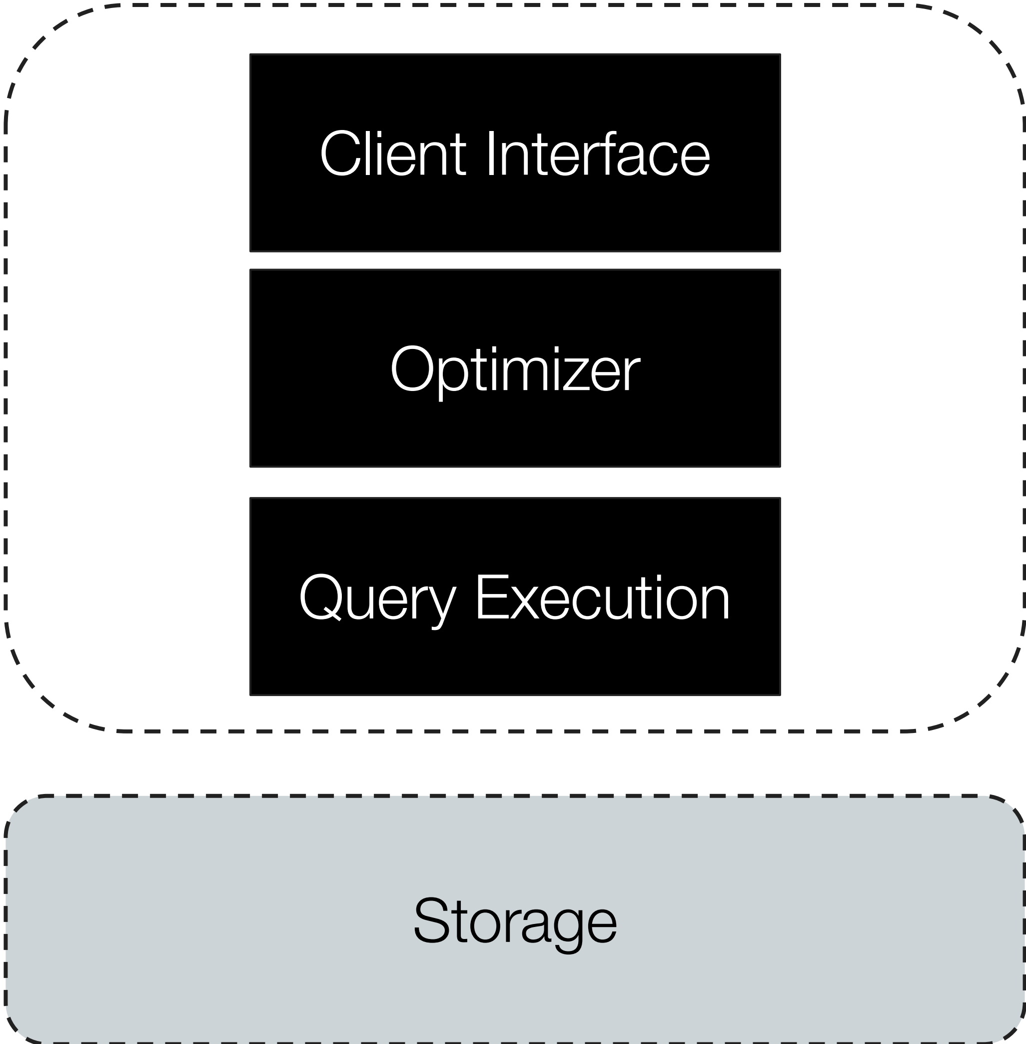
- Database Internals: Overview
  - OLAP and OLTP databases
  - Storage models
  - In-database optimizations

- Performance Pitfalls
  - Doing unnecessary work- ORMs, Absent or bad indexes, Layout
  - Read/Write amplification

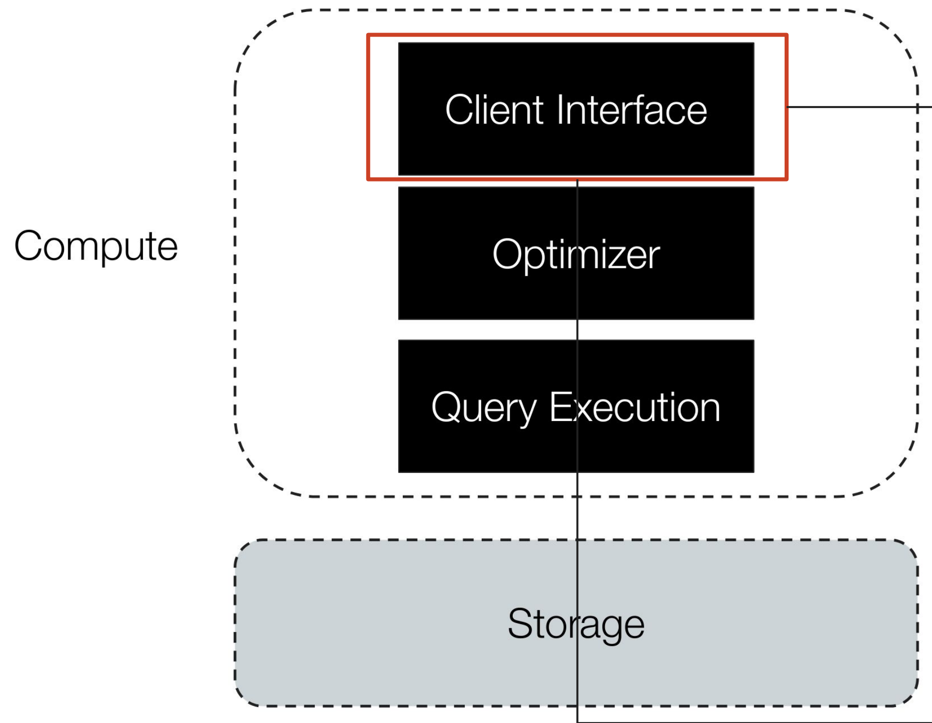
Components of a database

What *is* a database?

Compute



Interface




```
8   async function run() {
9       try {
10          await client.connect();
11
12          // set namespace
13          const database = client.db("sample_airbnb");
14          const coll = database.collection("listingsAndReviews");
15
16          // define pipeline
17          const agg = [
18              {
19                '$search': {
20                  'index': 'geo-json-tutorial',
21                  'compound': {
22                    'must': [
23                      {
24                        'geoWithin': {
25                          'geometry': {
26                            'type': 'Polygon',
27                            'coordinates': [
28
29
30
31
```

Filtering and retrieving data using Amazon S3 Select

PDF | RSS

Important

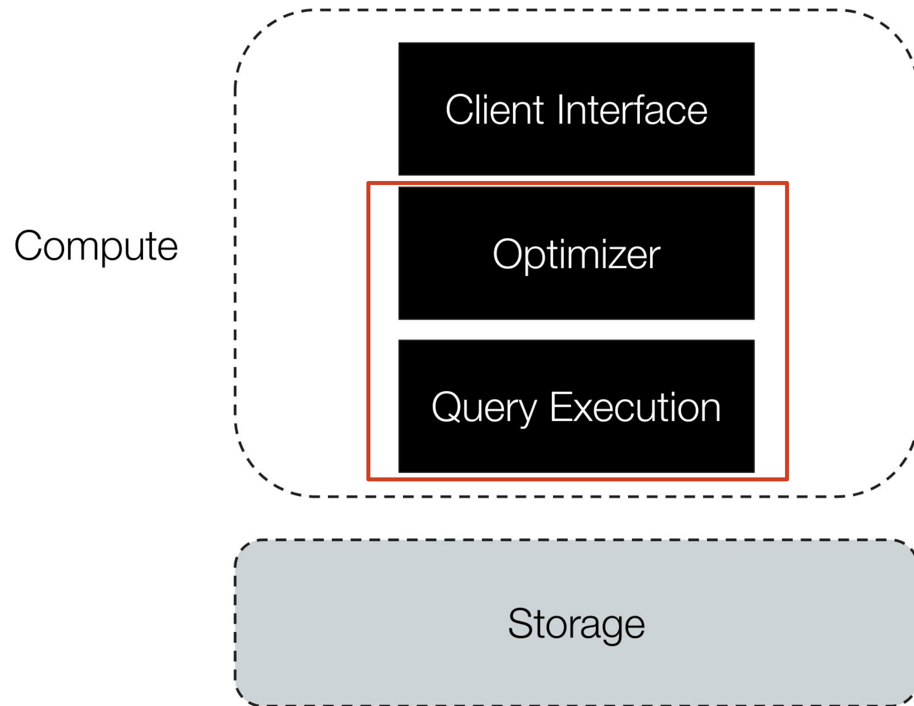
Amazon S3 Select is no longer available to new customers. Existing customers of Amazon S3 Select can continue to use the feature as usual.
Learn more

With Amazon S3 Select, you can use structured query language (SQL) statements to filter the contents of an Amazon S3 object and retrieve only the subset of data that you need. By using Amazon S3 Select to filter this data, you can reduce the amount of data that Amazon S3 transfers, which reduces the cost and latency to retrieve this data.

Amazon S3 Select only allows you to query one object at a time. It works on an object stored in CSV, JSON, or Apache Parquet format. It also works with an object that is compressed with GZIP or BZIP2 (for CSV and JSON objects only), and a server-side encrypted object. You can specify the format of the results as either CSV or JSON, and you can determine how the records in the result are delimited.

Interface can be anything!
JSON for MongoDB, or SQL on S3 using REST API

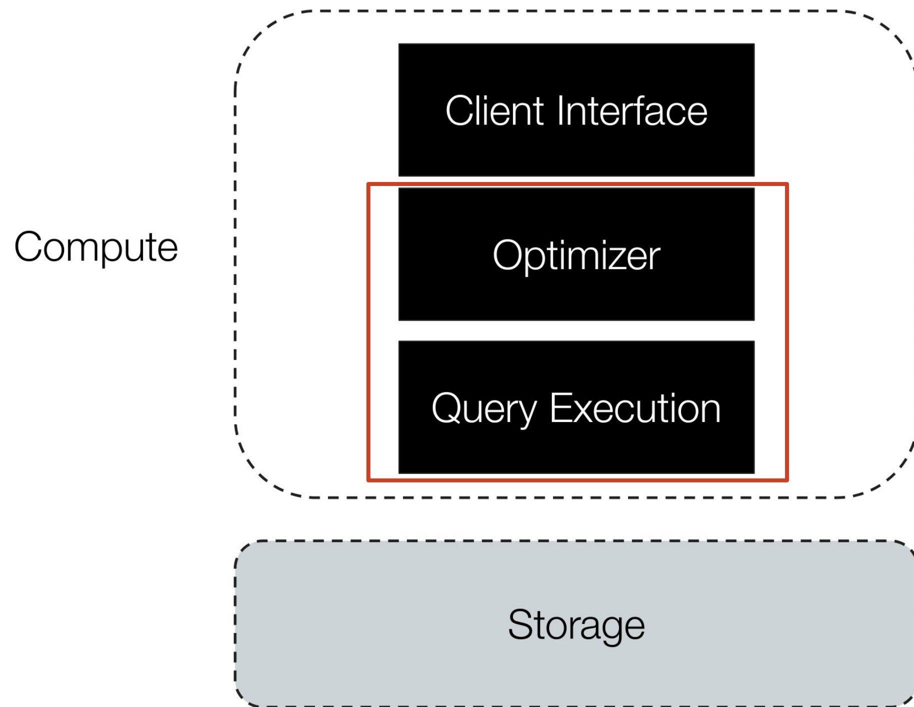
Optimizer and Query Execution



Optimizer is necessary for declarative client interfaces (like SQL), and it's job is to formulate an optimized **execution plan**

Query execution is the actual computation of the query result

Optimizer and Query Execution



Orca: A Modular Query Optimizer Architecture for Big Data

Mohamed A. Soliman*, Lyublena Antova*, Venkatesh Raghavan*, Amr El-Helw*,
Zhongxian Gu*, Entong Shen*, George C. Caragea*, Carlos Garcia-Alvarado*,
Foyzur Rahman*, Michalis Petropoulos*, Florian Waas‡,
Sivaramakrishnan Narayanan§, Konstantinos Krikellas†, Rhonda Baldwin*

* Pivotal Inc. Palo Alto, USA
‡ Datometry Inc. San Francisco, USA
† Google Inc. Mountain View, USA
§ Qubole Inc. Mountain View, USA

ABSTRACT

The performance of analytical query processing in data man-
agement systems depends primarily
the system's query optimizer. Increas
heightened interest in processing comp
have prompted Pivotal to build a new
In this paper we present the architec
query optimizer for all Pivotal data m
including Pivotal Greenplum Databas
Orca is a comprehensive development
art query optimization technology with
resulting in a modular and portable o
In addition to describing the overall
light several unique features and prese
parisons against other systems.

Categories and Subject Descri

H.2.4 [**Database Management**]:
cessing; Distributed databases

Despite a plethora of research in this area, most exist-
ing query optimizers in both commercial and open source
projects are still primarily based on technology dating back

Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources

Edmon Begoli
Oak Ridge National Laboratory (ORNL)
Oak Ridge, Tennessee, USA
begolie@ornl.gov

Jesús Camacho-Rodríguez
Hortonworks Inc.
Santa Clara, California, USA
jcamacho@hortonworks.com

Julian Hyde
Hortonworks Inc.
Santa Clara, California, USA
jhyde@hortonworks.com

Michael J. Mior
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
mmior@uwaterloo.ca

Daniel Lemire
University of Quebec (TELUQ)
Montreal, Quebec, Canada
lemire@gmail.com

ABSTRACT

Apache Calcite is a foundational software framework that provides query processing, optimization, and query language support to many popular open-source data processing systems such as Apache Hive, Apache Storm, Apache Flink, Druid, and MapD. The goal of this paper is to formally introduce Calcite to the broader research community, briefly present its history, and describe its architecture, features, functionality, and patterns for adoption. Calcite's architecture consists of a modular and extensible query optimizer with hundreds of built-in optimization rules, a query processor capable of processing a variety of query languages, an adapter architecture designed for extensibility, and support for heterogeneous data models and stores (relational, semi-structured, streaming, and geospatial). This flexible, embeddable, and extensible architecture is what makes Calcite an attractive choice for adoption in big-data frameworks. It is an active project that continues to introduce support for the new types of data sources, query languages, and approaches to query processing and optimization.

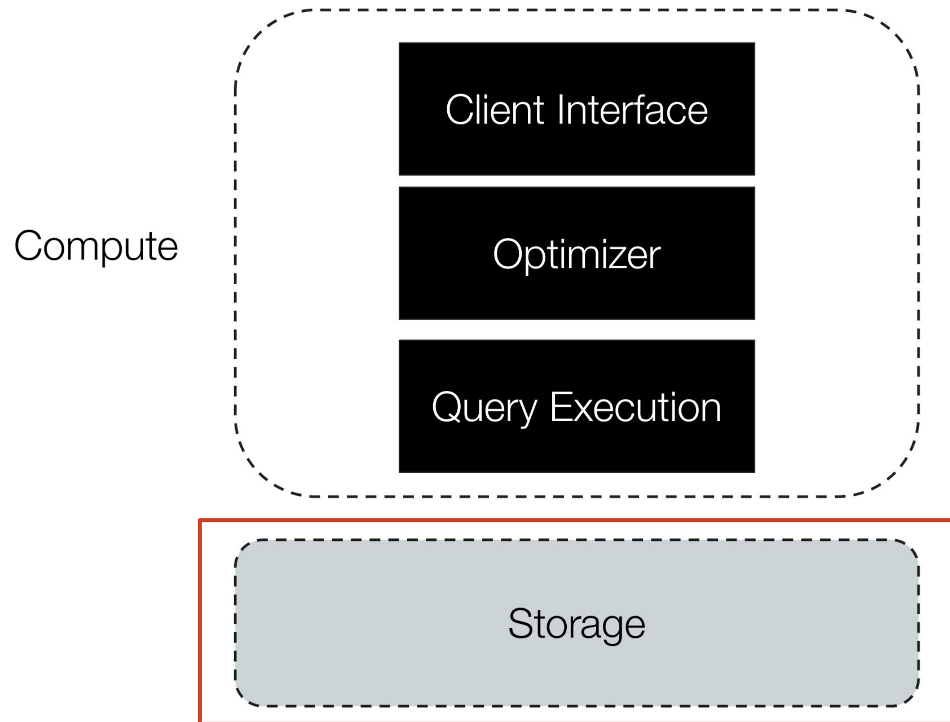
Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183713.3190662

1 INTRODUCTION

Following the seminal System R, conventional relational database engines dominated the data processing landscape. Yet, as far back as 2005, Stonebraker and Çetintemel [49] predicted that we would see the rise a collection of specialized engines such as column stores, stream processing engines, text search engines, and so forth. They argued that specialized engines can offer more cost-effective performance and that they would bring the end of the "one size fits all" paradigm. Their vision seems today more relevant than ever. Indeed, many specialized open-source data systems have since become popular such as Storm [50] and Flink [16] (stream processing), Elasticsearch [15] (text search), Apache Spark [47], Druid [14], etc.

As organizations have invested in data processing systems tai-

Storage



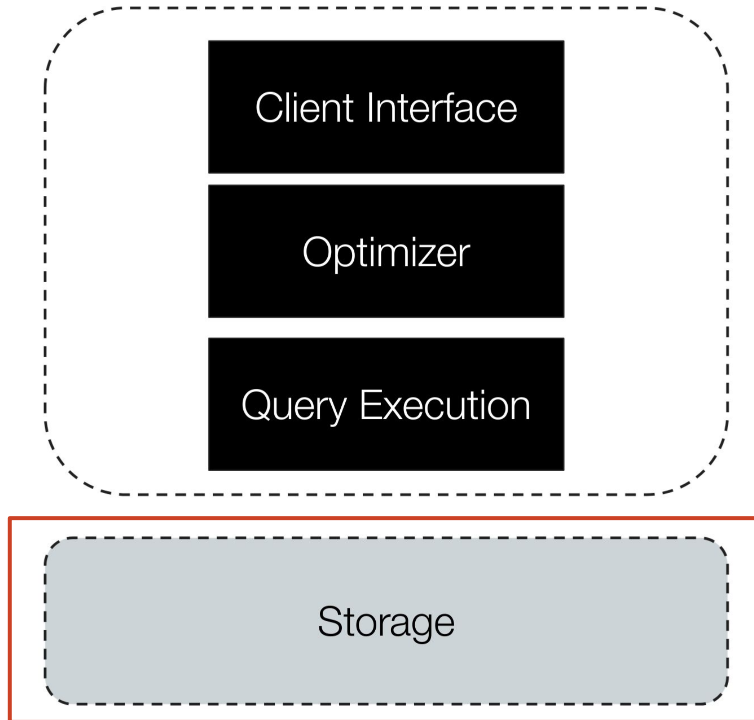
Recent times: lot of innovation in the domain of storage

Some systems leverage object stores for durable storage

Others are scaling storage by distributing it

Storage

Compute



Today's class

- Why care about a database?

- Components of a database
  - Compute (query) and Storage

- **Database Internals: Overview**
  - **OLAP and OLTP databases**
  - **Storage models**
  - **In-database optimizations**

- Performance Pitfalls
  - Doing unnecessary work- ORMs, Absent or bad indexes, Layout
  - Read/Write amplification

Database Internals

The OLAP/OLTP Classification

- Database implementation depends the **application it is geared towards** (i.e. the kind of guarantees, and queries to be supported)

- **storage and compute** are implemented (and optimized) for the same

- A very popular way to divide is between **transactional databases (OLTP), and analytics databases (OLAP)**

The OLTP Database

Online Transactional Processing (OLTP) databases are used to implement “transactions” to support application logic

- In-general they allow all operations like **READ, INSERT, UPDATE**, and **DELETE**
- **Strong guarantees (like ACID)** are usually a necessity for correctness
- Traditionally most transactional databases have been **relational** (i.e. SQL based), for example PostgreSQL, MySQL

The OLAP Database

Online Analytical Processing (OLAP) databases are used to answer analytical queries over data

- OLAP databases are optimized for **large scale READs**, and aggregation based queries

- In general strong **guarantees are not necessary**, especially not for “ALL” operations. This relaxation can give very strong performance upsides!

- Interestingly, **SQL is also a commonly used** client interface for these, (eg, Clickhouse)

Before we go into the internals we will look at PostgreSQL as an example of how a database is organised

Following slides will cover background on Postgres

PostgreSQL: Broad Overview

A PostgreSQL server/cluster manages data in **multiple "databases"**. Each database **contains tables, views and other objects**. This is analogous to hierarchical organization in a file system.

PosgtreSQL contains 3 identical databases on startup

- **template0** is used for cases like restoring data from a logical backup or creating a database with a different encoding; it must never be modified
- **template1** serves as a template for all the other databases that a user can create in the cluster
- **postgres** is a regular database that you can use at your discretion

PostgreSQL: System Catalogs

- **Metadata of all cluster objects** (such as tables, indexes, data types, or functions) is stored in tables that belong to the system catalog.

- Each **database has its own set of tables** (and views) that describe the objects of this database.

- Several **system catalog tables are common to the whole cluster** (technically, a dummy database with a zero is used), but can be accessed from all of them.

PostgreSQL: System Catalogs

- The system catalog can be **viewed using regular queries**, while all **modifications** in it are performed by **DDL commands**

- The psql client also offers commands for this

- Names of all system catalog tables begin with `pg_`, like in `pg_database`

- Column names start with a three-letter prefix that usually corresponds to the table name, like in `datname`

PostgreSQL: Schemas (Namespaces)

Schemas are **namespaces that store all objects of a database**. Apart from user schemas, PostgreSQL offers several predefined ones:

`public` is the default schema for user objects unless other settings are specified.

`pg_catalog` is used for system catalog tables.
(information_schema provides an alternative view for the system catalog)

`pg_toast` is used for objects related to TOAST

`pg_temp` comprises temporary tables

Database Internals: Storage Models!

(slide illustrations- CMU Advanced Database Systems)

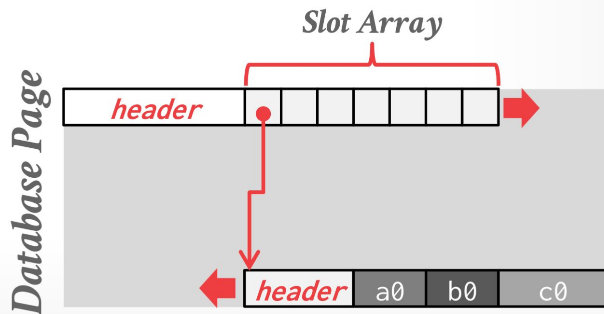
N-ary storage model (Row based storage)

- The DBMS stores (almost) all the **attributes for a single tuple contiguously** in a **single page**.
- Ideal for **OLTP workloads** where transactions tend to access individual entities and **insert-heavy workloads**.
  → Use the tuple-at-a-time iterator processing model.
- NSM database page sizes are typically some constant multiple of 4 KB hardware pages.
  → Example: Oracle (4 KB), Postgres (8 KB), MySQL (16 KB)

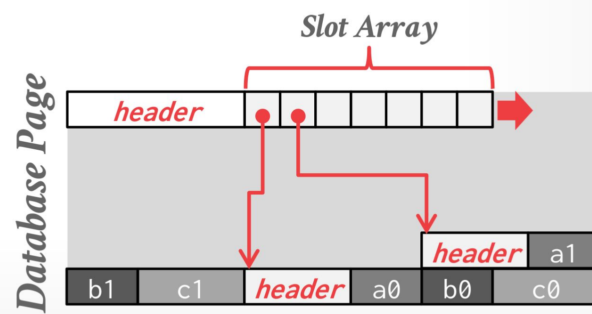
N-ary storage model (NSM): Physical Layout

A disk-oriented NSM system stores a tuple's **fixed-length and variable-length attributes contiguously** in a single slotted page. The tuple's **record id (page#, slot#)** is how the DBMS **uniquely identifies** a physical tuple.

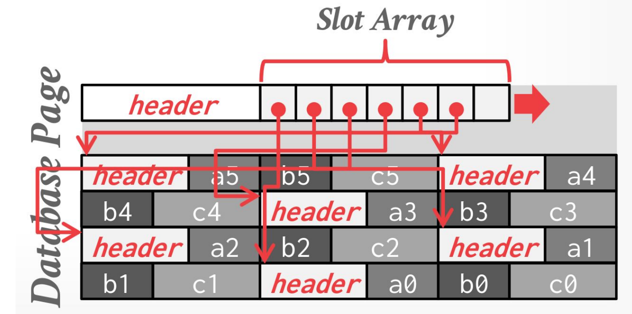
	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5

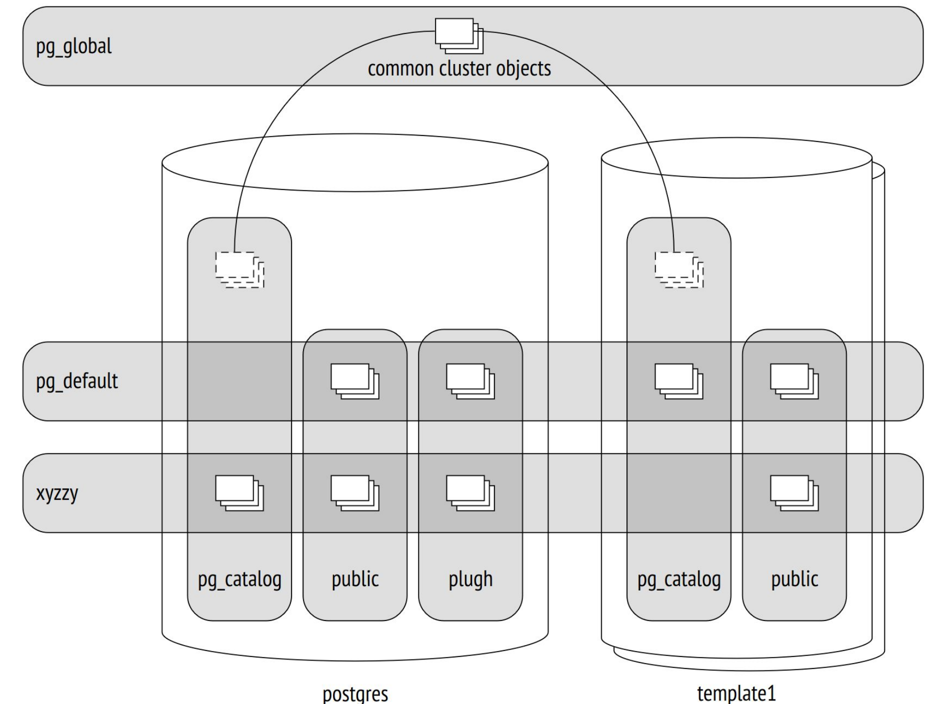


N-ary storage in PostgreSQL

- Databases and schemas determine logical distribution of objects, while tablespaces define physical data layout.

- A tablespace is implemented as a directory in a file system.

- You can distribute your data between tablespaces in such a way that archive data is stored on slow disks, while the data that is being actively updated goes to fast disks.



Tablespaces in PostgreSQL

- One and the same tablespace can be used by different databases, and each database can store data in several tablespaces.

- It means that logical structure and physical data layout do not depend on each other.

- Each database has the so-called *default tablespace*. All database objects are created in this tablespace unless another location is specified. System catalog objects related to this database are also stored there.

We have seen the **tablespace organization at the directory level**. Now we will look at how **tables, and indexes are stored using files**!

File level layout

- Each table and index is stored in a separate file. For ordinary relations, these files are named after the table or index's file-node number, which can be found in `pg_class.relfilenode`

- Each table and index has a free space map, which stores information about free space available in the relation. The free space map is stored in a file named with the filenode number plus the suffix `_fsm`

- Tables also have a visibility map, stored in a fork with the suffix `_vm`, to track which pages are known to have no dead tuples. The visibility map

Now we will look at the **page layout**!

Tuple arrangement in file pages

- By design, Postgres allows multiple tuples in a page, but **one tuple must not exceed a page**! (we look at how a page is arranged shortly)

- To accommodate cases like this, Postgres uses a mechanism called **TOAST (The Oversized Attributes Storage Technique)**

- **TOAST Strategies**:
  - Move long attribute values into a separate service table, having sliced them into smaller “toasts.”
  - compress a long value in such a way that the row fits the page.
  - Or you can do both: first compress the value, and then slice and move

TOAST and Potential Pitfalls

- If the main table contains potentially long attributes, a **separate table is created for it right away**, one for all the attributes.

- For example, if a table has a column of the **numeric or text type**, a table will be created even if this column will never store any long values.

- For **indexes, the mechanism can offer only compression**; moving long attributes into a separate table is not supported. It **limits the size of the keys** that can be indexed

- Simplest way to **review the used strategies** is to run the `\d+` command in `psql`

Page level layout

Item	Description
PageHeaderData	24 bytes long. Contains general information about the page, including free space pointers.
ItemIdData	Array of item identifiers pointing to the actual items. Each entry is an (offset,length) pair. 4 bytes per item.
Free space	The unallocated space. New item identifiers are allocated from the start of this area, new items from the end.
Items	The actual items themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.

Field	Type	Length	Description
pd_lsn	PageXLogRecPtr	8 bytes	LSN: next byte after last byte of WAL record for last change to this page
pd_checksum	uint16	2 bytes	Page checksum
pd_flags	uint16	2 bytes	Flag bits
pd_lower	LocationIndex	2 bytes	Offset to start of free space
pd_upper	LocationIndex	2 bytes	Offset to end of free space
pd_special	LocationIndex	2 bytes	Offset to start of special space
pd_pagesize_version	uint16	2 bytes	Page size and layout version number information
pd_prune_xid	TransactionId	4 bytes	Oldest unpruned XMAX on page, or zero if none

The first table details the page layout

And the second table details the page metadata stored

And that is how N-ary storage works!

Rest is essentially ACID guarantees, and indexes like B-Trees you have studied in your Database course :)

N-ary storage model: Advantages

- Fast INSERT, UPDATES, and DELETES to support transactions
- Good for queries that need the entire tuple
- Allows for strong guarantees (required for OLTP) on the level of a single record
- Can use index-oriented physical storage for clustering to improve performance

N-ary storage model: Disadvantages

- Not good for scanning large portions of the table and/or a subset of the attributes (eg, aggregating a column).

- Terrible memory locality in access patterns. This is fine as long as we don't need to scale reads.

- Not ideal for compression because of multiple value domains within a single page. Compression algorithms work (much) better when type information is available (and different algorithms can be used for different columns)

Decomposition Storage Model (DSM, Column based)

- The DBMS stores a single attribute (column) for all tuples contiguously in a block of data.

- Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attributes.
  → Use a batched vectorized processing model.

- File sizes are larger (100s of MBs), but it may still organize tuples within the file into smaller groups.

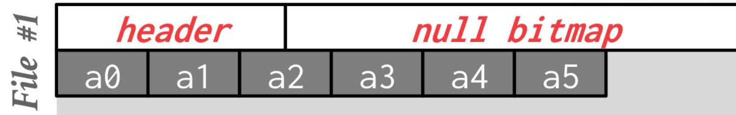
Decomposition Storage Model: Physical Layout

- Store **attributes and metadata (e.g., nulls) in separate arrays** of fixed- length values.
  → Most systems identify unique physical tuples using offsets into these arrays.

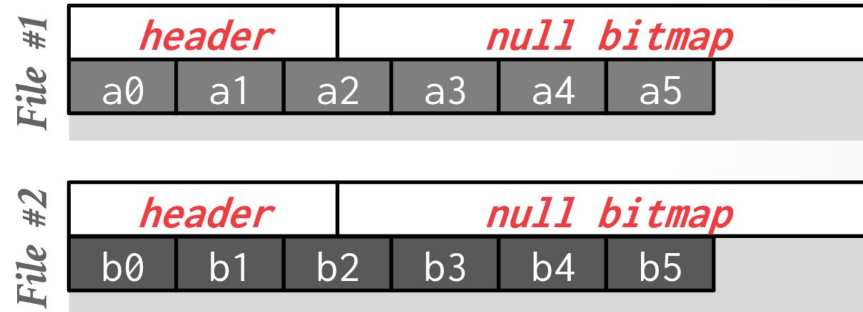
- To handle **variable-length values** we can maintain a separate file per attribute with a dedicated header area for metadata about entire column.

Decomposition Storage Model: Physical Layout

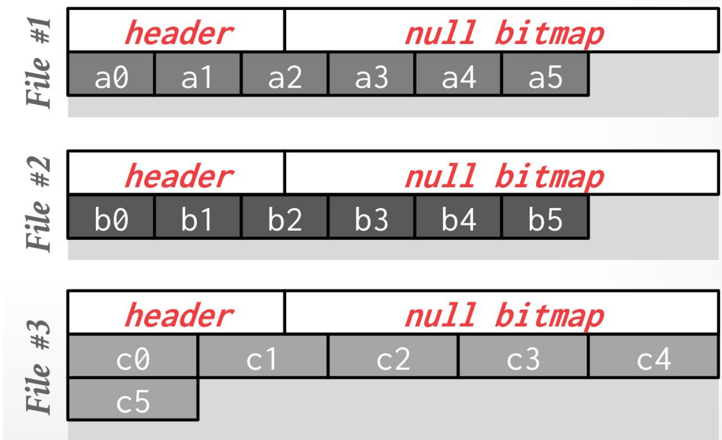
	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



	Col A	Col B	Col C
Row #0	a0	b0	c0
Row #1	a1	b1	c1
Row #2	a2	b2	c2
Row #3	a3	b3	c3
Row #4	a4	b4	c4
Row #5	a5	b5	c5



Decomposition Storage Model: Tuple Identification

- Choice #1: Fixed-length Offsets
  → Each value is the same length for an attribute.

- Choice #2: Embedded Tuple Ids
  → Each value is stored with its tuple id in a column.

Offsets

	A	B	C	D
0				
1				
2				
3				

Embedded Ids

	A		B		C		D
0		0		0		0	
1		1		1		1	
2		2		2		2	
3		3		3		3	

Decomposition Storage Model: Advantages/Disadvantages

Advantages

- Reduces the amount **wasted I/O per query** because the DBMS only reads the data that it needs.
- Faster **query processing** because of increased locality and cached data reuse.
- Better **data compression** (more on this later)

Disadvantages

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

Just for reference: PAX Storage Model

- Partition Attributes Across (PAX) is a **hybrid storage model** that vertically partitions attributes within a database page.
  → This is what **Paraquet and Orc** use.

- The goal is to get the benefit of **faster processing** on columnar storage while retaining the spatial locality benefits of row storage.

Database Internals: Indexes, and other optimizations

(slide illustrations- CMU Advanced Database Systems)

Storage Optimizations

- We can of course optimize storage throughput/latency/reliability by **distributing a database**

- Or by introducing external **caching/query coalescing services** (Redis, or response caching or middlewares).

- We will cover this in later lectures! For now we discuss **in-database optimizations** that are present in many databases

In-database Design-choices and Optimizations

- Buffer management *(we are skipping this)*
- **Transparent Huge-pages**
- Numeric representation
- NULL representation
- OLAP Indexes

Huge-pages: Motivation

- **TLBs are pretty small because they need to be fast**.

AMD's Zen 4 Microarchitecture, which first shipped in September 2022, has a first level data TLB with **72 entries, and a second level TLB with 3072 entries**.

- That means **avoiding page-table lookups is hard**!

In Zen 4 when an application's **working set is larger than approximately 4 kiB × 3072 = 12 MiB**, some memory accesses will require page table lookups,

Huge-pages: Motivation

- **TLBs are pretty small because they need to be fast**.

- That means **avoiding page-table lookups is hard**!

Larger virtual memory page sizes (aka huge pages) can reduce page mapping overhead substantially.

Transparent Huge-pages

Instead of always allocating memory in 4 KB pages, Linux supports creating larger pages (2MB to 1GB)
→ Each page must be a contiguous blocks of memory.
→ Greatly reduces the # of TLB entries

With THP, the OS reorganizes pages in the background to keep things compact.
→ Split larger pages into smaller pages.
→ Combine smaller pages into larger pages.
→ Can cause the DBMS process to stall on memory access.

Issues with Transparent Huge-pages

Transparency has a cost! Given that a page has to be in **contiguous memory, creating huge-pages** requires memory defragmentation, this has added latency and processing overhead

Historically, every DBMS advises you to disable this THP on Linux:
→ Oracle, SingleStore, NuoDB, MongoDB, Sybase, TiDB.
→ Vertica says to enable THP only for newer Linux distros.

But in many cases huge pages (transparent or otherwise) have improved performance, so always measure!

In-database Design-choices and Optimizations

- Buffer management *(we are skipping this)*
- Transparent Huge-pages
- **Numeric representation**
- NULL representation
- OLAP Indexes

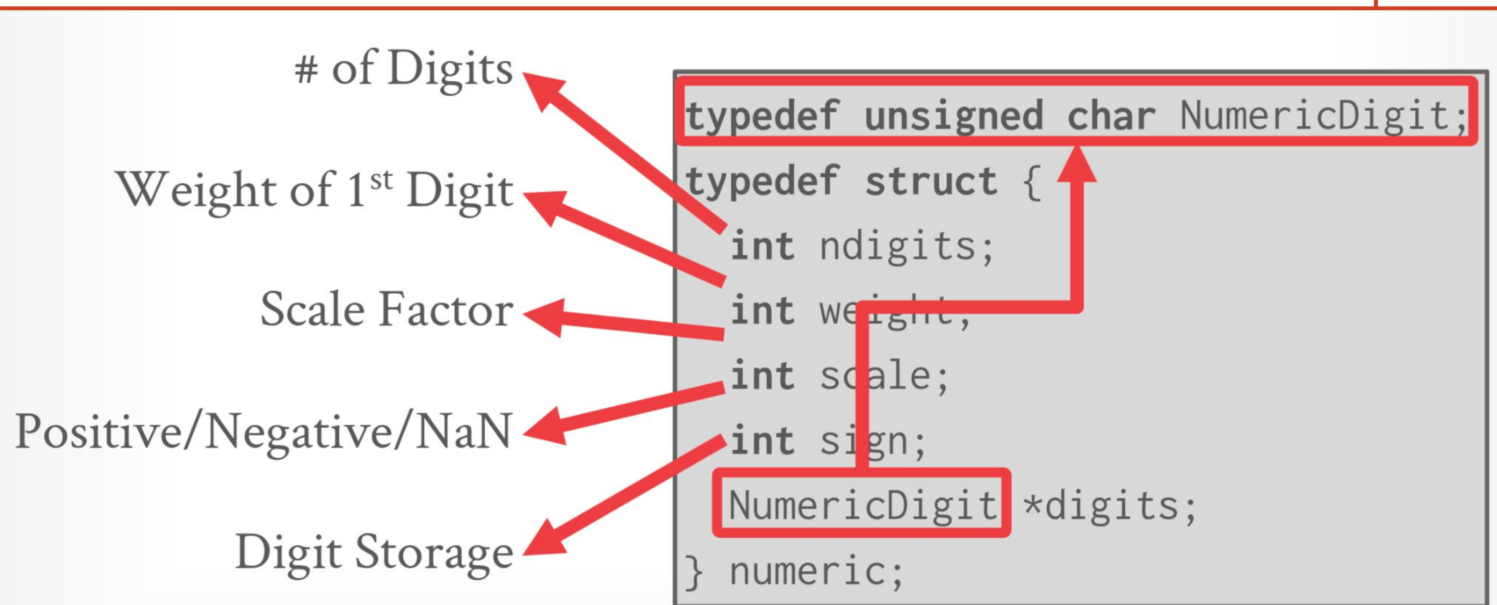
Numeric representation: Variable Precision

- Inexact, variable-precision numeric type that uses the "native" C/C++ types. Store directly as specified by IEEE-754.
  → Example: FLOAT, REAL/DOUBLE

- These types are typically faster than fixed precision numbers because CPU ISA's (Xeon, Arm) have instructions / registers to support them. But they do not guarantee exact values

Numeric representation: Fixed Precision

Arbitrary precision is inefficient!
Definition of `Postgres numeric` is below
To the side you see (part of) code to add 2 `Postgres numerics`!



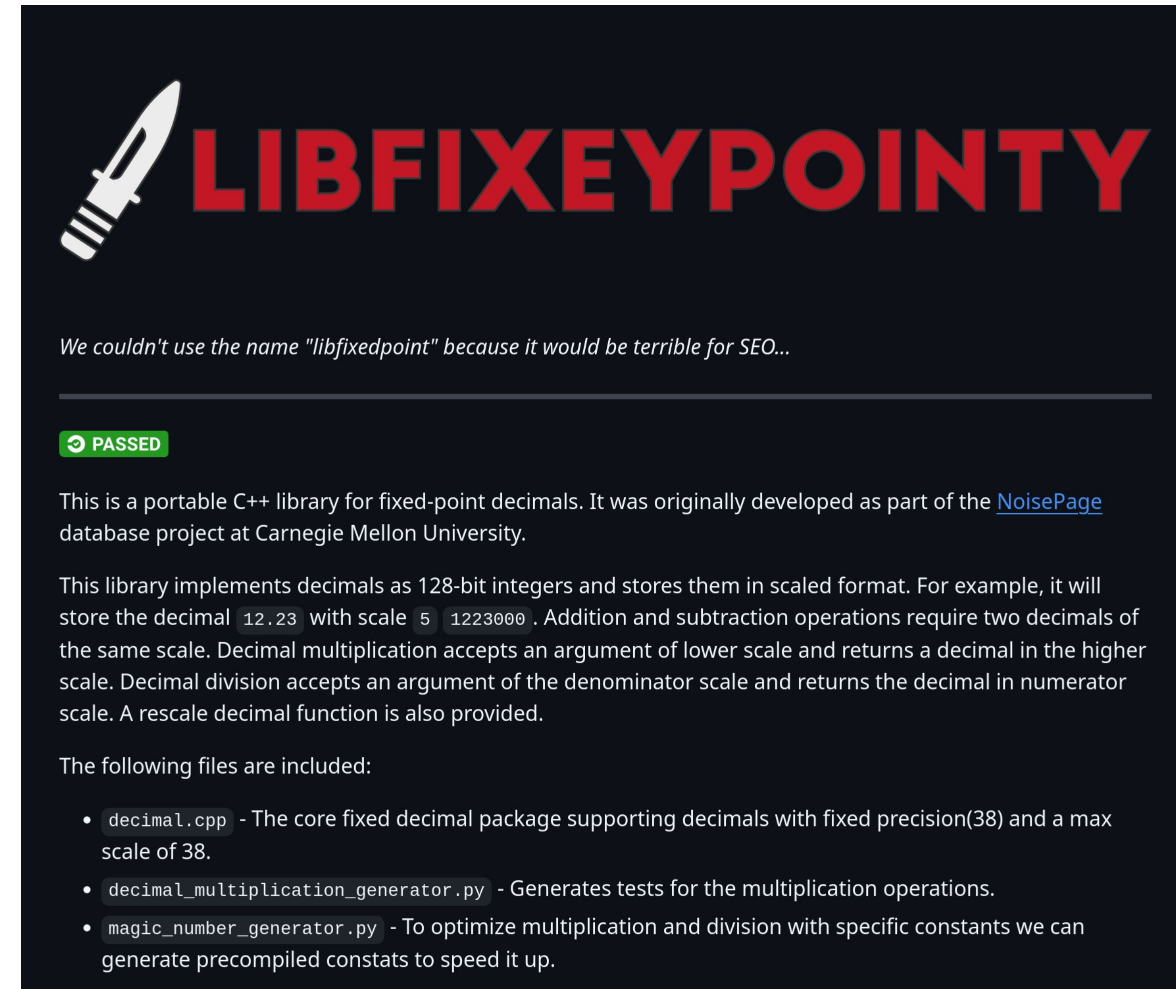

```c
/* ----------
 * add_var() -
 *
 *  Full version of add functionality on variable level (handling signs).
 *  result might point to one of the operands too without danger.
 * ----------
 */
int
PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /*
     * Decide on the signs of the two variables what to do
     */
    if (var1->sign == NUMERIC_POS)
    {
        if (var2->sign == NUMERIC_POS)
        {
            /*
             * Both are positive result = +(ABS(var1) + ABS(var2))
             */
            if (add_abs(var1, var2, result) != 0)
                return -1;
            result->sign = NUMERIC_POS;
        }
        else
        {
            /*
             * var1 is positive, var2 is negative Must compare absolute values
             */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                    /* ----------
                     * ABS(var1) == ABS(var2)
                     * result = ZERO
                     * ----------
                     */
                    zero_var(result);
                    result->rscale = Max(var1->rscale, var2->rscale);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;

                case 1:
                    /* ----------
                     * ABS(var1) > ABS(var2)
                     * result = +(ABS(var1) - ABS(var2))
                     * ----------
                     */
                    if (sub_abs(var1, var2, result) != 0)
                        return -1;
                    result->sign = NUMERIC_POS;
                    break;

                case -1:
                    /* ----------
                     * ABS(var1) < ABS(var2)
                     * result = -(ABS(var2) - ABS(var1))
                     * ----------
                     */
                    if (sub_abs(var2, var1, result) != 0)
                        return -1;
                    result->sign = NUMERIC_NEG;
                    break;
            }
```

Numeric representation: Fixed Precision

- Use (only) when rounding errors are literally unacceptable! **Use variable precision (floating point) instead**

- Implementation generally uses a variable length **binary representation with additional metadata**

- People have tried optimizations!



In-database Design-choices and Optimizations

- Buffer management *(we are skipping this)*
- Transparent Huge-pages
- Numeric representation
- **NULL representation**
- OLAP Indexes

Representing NULLs!

Choice #1: Special Values

→ Designate a value to represent NULL for a data type (e.g., INT32_MIN).

Choice #2: Null Column Bitmap Header

→ Store a bitmap in a centralized header that specifies what attributes are null.

Choice #3: Per Attribute Null Flag

→ Store a flag that marks that a value is null.

→ Must use more space than just a single bit because this messes up with word alignment.