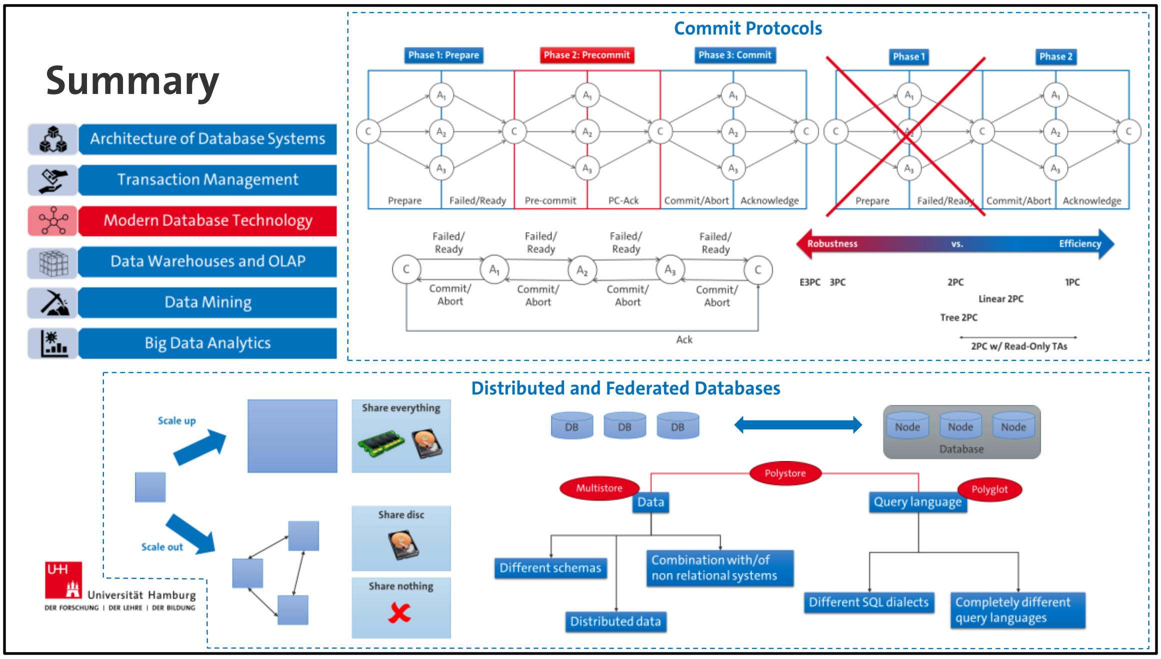

Summary
Architecture of Database Systems
Transaction Management
Modern Database Technology
Data Warehouses and OLAP
Data Mining
Big Data Analytics
Commit Protocols
Phase 1: Prepare
Phase 2: Precommit
Phase 3: Commit
Prepare
Failed/Ready
Pre-commit
PC-Ack
Commit/Abort
Acknowledge
Phase 1
Phase 2
Prepare
Failed/Ready
Commit/Abort
Acknowledge
Failed/ Ready
Commit/ Abort
Ack
Robustness
vs.
Efficiency
E3PC
3PC
2PC
1PC
Linear 2PC
Tree 2PC
2PC w/ Read-Only TAs
Distributed and Federated Databases
Scale up
Scale out
Share everything
Share disc
Share nothing
DB
Node
Database
Multistore
Polystore
Polyglot
Data
Query language
Different schemas
Combination with/of non relational systems
Distributed data
Different SQL dialects
Completely different query languages
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Course Outline

- Architecture of Database Systems
- Transaction Management
- Modern Database Technology
  - Distributed Systems
  - **Optimizations in RDBMS**
  - NoSQL
- Data Warehouses and OLAP
- Data Mining
- Big Data Analytics

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Storage Layouts

2 main layouts to store your table

MensaMeals

Meal	Price
Pizza	6,50
Pasta	4,90
Pie	1,20
Potato Salad	5,80
Pannfisch	7,90

Row-Store (tuple-wise)

This is what your traditional relational SQL database does

Pizza	6,50	Pasta	4,90	Pie	1,20	Potato Salad	5,80	Pann-fisch	7,90

Memory address →

Column-Store (attribute-wise)

This is what all (not so traditional) column-oriented databases do

Pizza	Pasta	Pie	Potato Salad	Pann-fisch					

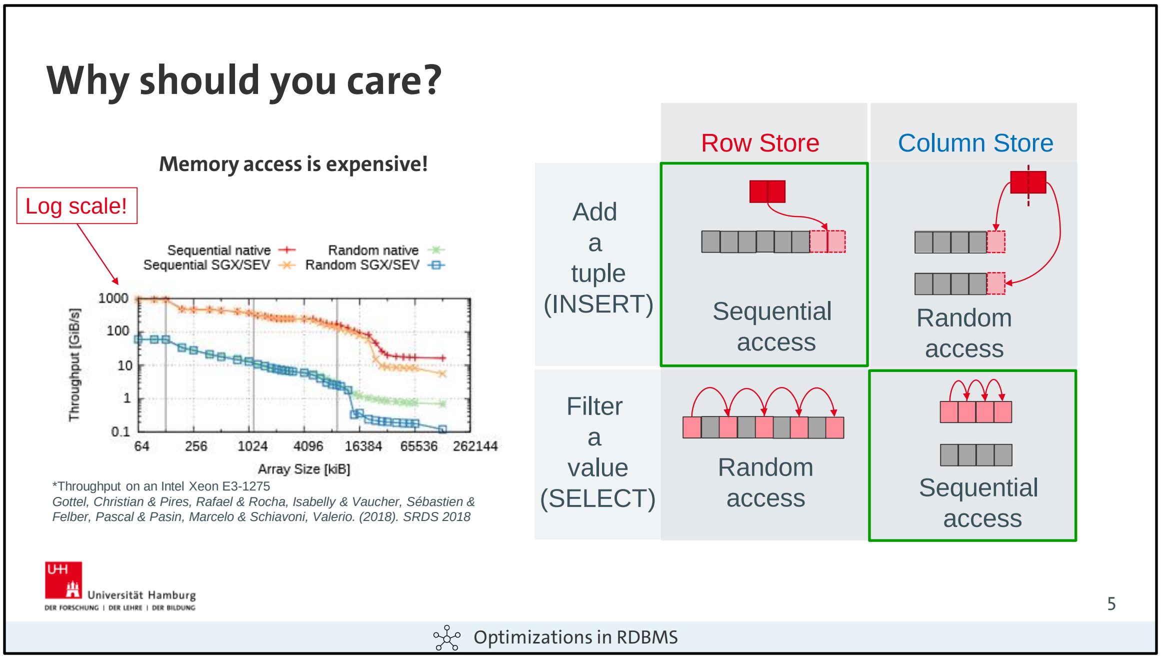
Memory address →

6,50	4,90	1,20	5,80	7,90					

Memory address →

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Relations are usually illustrated as tables
→ This tells us nothing about the storage layout
(cf. a matrix that can be stored differently → row- or column-major)

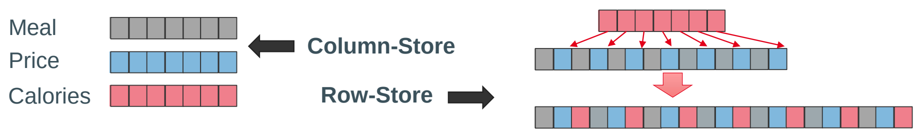


Your ideal layout depends on your use-case.
Different systems use different layouts, so choose wisely!

NoSQL and column-oriented DBs: Frequent misunderstandings

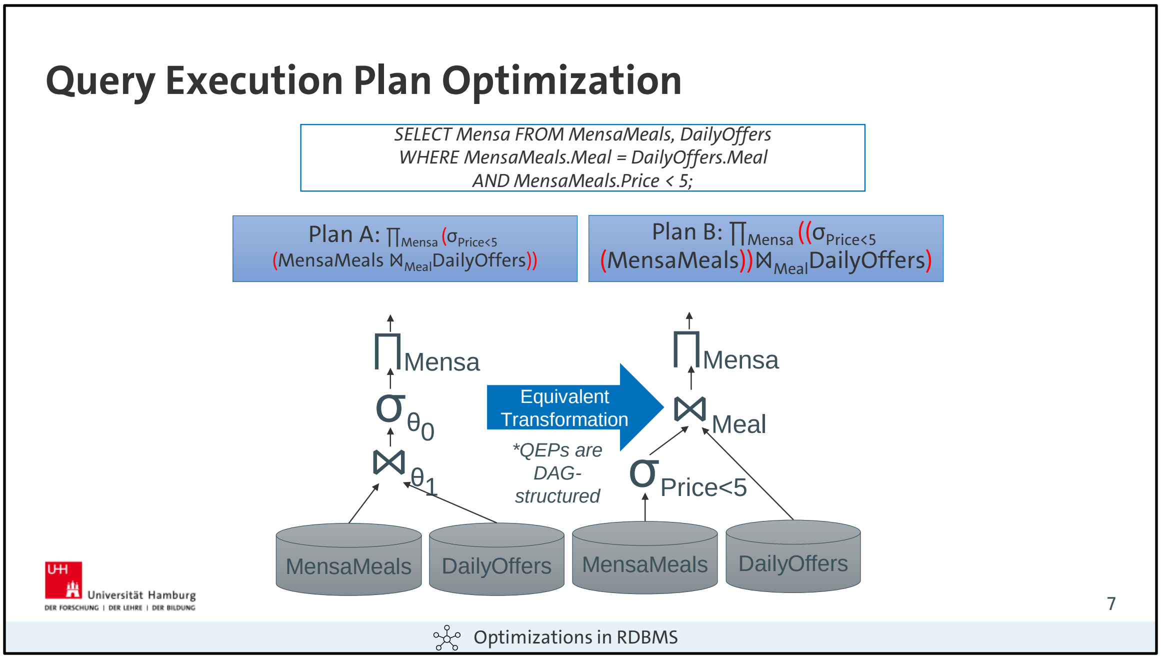
Wide-column DBs (NoSQL) = column-stores (SQL)

ALTER TABLE *MensaMeals* ADD *Calories* INT NULL;



Remember what we just learned about random memory access

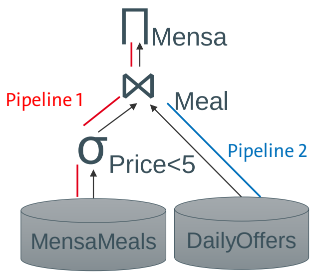
- NoSQL stands for **N**ot **o**nly **SQL**
- Wide-column DBs (NoSQL) and column-stores (SQL) are not the same, but both often referenced as column-oriented
  → We will use it to reference column-stores
- Usually, column-oriented databases can be queried using SQL and allow the definition of relations
  → Convenience of SQL, and performance and flexibility of column-stores
  → Example: Fast and easy addition/deletion of attributes



- Database Systems use a relational algebra for internal representation
- Optimizers try to automatically find the most efficient sequence of operators
  → Conventional approach: Reduce data as early and as cheap as possible
  → Tool: Cardinality/Selectivity estimation
- The chosen sequence of operators is the final Query Execution Plan (QEP)

- See 1st lecture of the semester for further reading

Operator Fusion



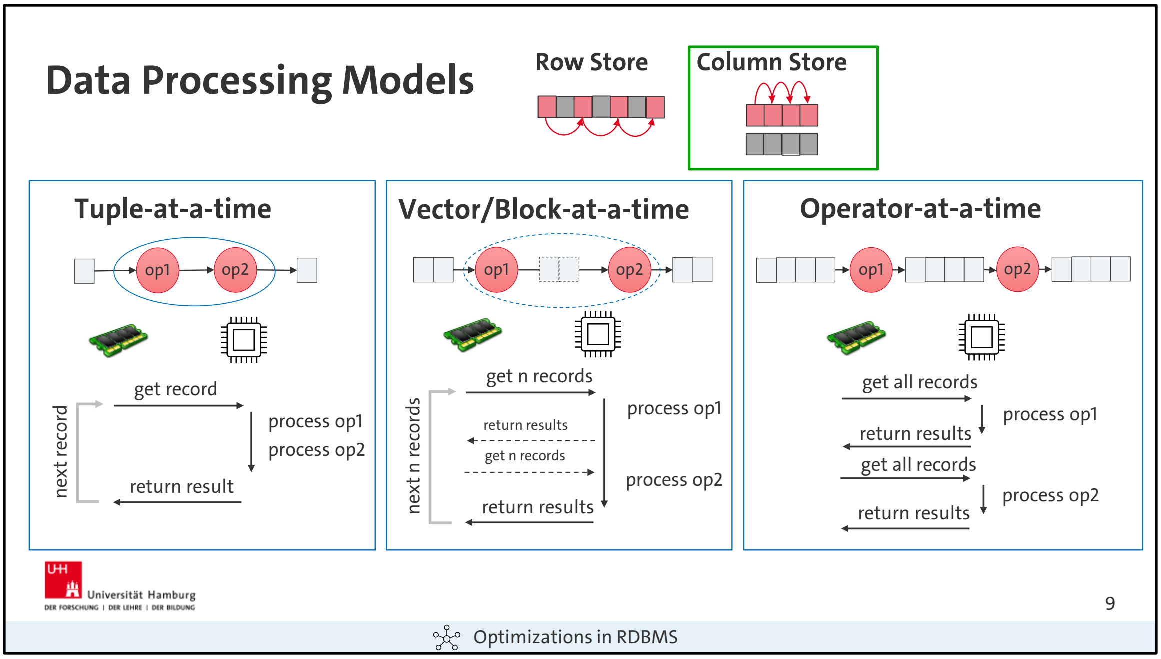
Tuples processed along the operators of a pipeline
→ Stay in the CPU registers while they are processed
→ No additional memory access

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Step 1: Optimizer identifies pipeline breakers, i.e. operators that must materialize (intermediate) results
Step 2: Query is compiled such that there is one loop for each pipeline
Step 3: Values/individual tuples are loaded into CPU registers
Step 4: Tuples are processed along their pipeline (pushed to the parent operators) without being materialized until the next pipeline breaker

Further Reading

Menon, Prashanth, Todd C. Mowry, and Andrew Pavlo. "Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last." *Proceedings of the VLDB Endowment* 11.1 (2017): 1-13.



For parallel or pipelined execution, data must be split

Tuple-at-a-time

- Intermediate tuples not stored, but passed directly to next operator
  →Operators can be fused
- Limited applicability of other optimizations, e.g. prefetching, vectorization, compression,...

Vector/Block-at-a-time

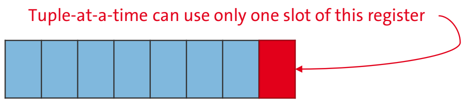
- A part (vector/block) of the column processed at once
  →Operator fusion only for small blocks
- Trade-off between operator fusion and memory access performance

Operator-at-a-time

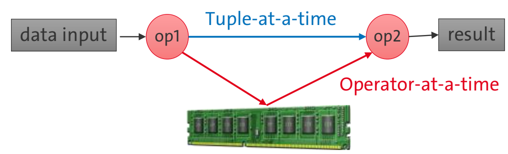
- Whole operator (all elements of the column) processed at once
- Intermediates materialized
  →No operator fusion, only coarse-grained parallelization
- High potential for optimization of memory reads

Why should I care?

- Different optimizations work with different processing models
- Your hardware limits your optimization space



Vector registers can hold multiple values which are then processed by the same instruction, e.g., up to 8 64-bit values with AVX512



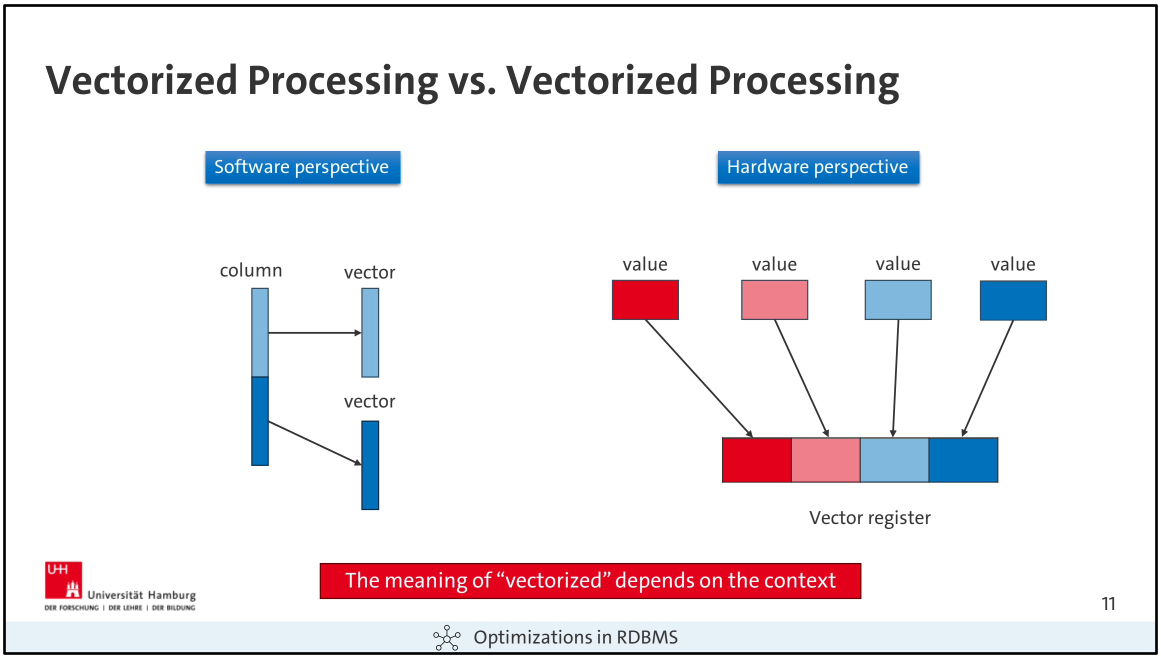
Intermediate results are usually larger than the L1 cache
→ Must be stored in a larger (and slower) part of the memory hierarchy before applying the next operator

Example A: You have a recent intel server with the AVX512 instruction set for vectorization (under linux, *lscpu* tells you if you have it; no root required)

➔ A system which implements only tuple-at-a-time is not able to use this instruction set

Example B: You do not have much main memory and writing to it is slow

➔ Materializing your intermediates becomes a bottleneck and might not work at all with operator-at-a-time or large blocks (block-at-a-time)



<u>Software perspective</u>
Data is divided into vectors or blocks and processed vector-wise → Vector-at-a-time
<u>Hardware perspective</u>
A single register (“vector register”) can hold multiple values which are then processed at the same time → Single Instruction Multiple Data (SIMD)

- A vector in vector-at-a-time processing can be as small as a vector register, but can also be significantly larger
- Both approaches can be used at the same time

Exercise Optimizations

- Assume a table *t(a1 SMALLINT, a2 FLOAT)* that has $20*10^9$ records
- The table resides in main memory
- Assume further that we process all queries on a system with two 64-bit memory channels and a maximum memory frequency of 2GHz
- How much time does it take **at least** to copy all values for the following query from main memory to L1 cache if a column store + vector-at-a-time is used?:

  SELECT a1 FROM t WHERE a1 < 5;
- How much time does it take for a row store + operator-at-a-time?

smallint → 16 bit
int → 64 bit (on most systems)

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

100bn smallint values = $100*10^9*2$ / 1024^3 = 186 GB

Ignored here (among others):
- NOPs
- Computation of the operators
- Computation of addresses
- Materialization of the (intermediate) results

Exercise Optimizations

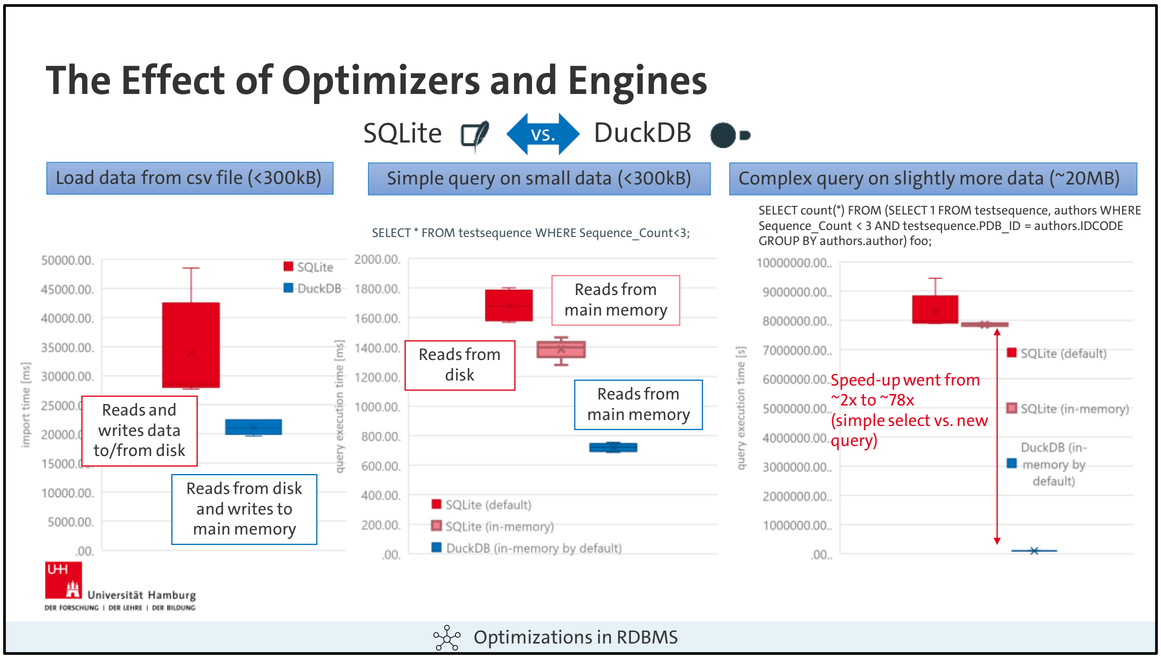
- Assume a table *t(a1 SMALLINT, a2 INT)* that has $20*10^9$ records
- The table resides in main memory
- Assume further that we process all queries on a system with two 64-bit memory channels and a maximum memory frequency of 2GHz
- How much time does it take **at least** to copy all values for the following query from main memory to L1 cache if a row store + operator-at-a-time + operator fusion is used?:

  SELECT a1+a2 FROM t WHERE a1 < 5;
- How much time does it take without operator fusion?
- How much time does it take for a column store + vector-at-a-time (no operator fusion)?

smallint → 16 bit
int → 64 bit (on most systems)

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

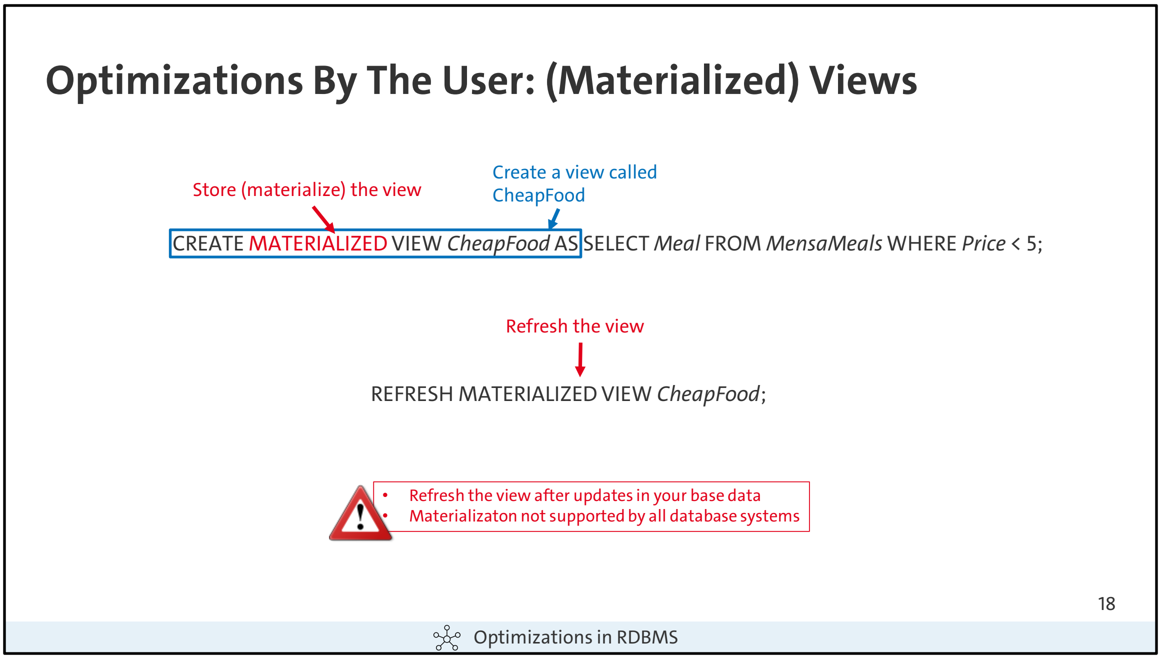
100bn smallint values = $100*10^9*2 / 1024^3$ = 186 GB



Data was taken from the Protein Data Bank (wwwpdb.org, accessed in 2022)

SQLite
- Row-Store
- Disc-centric
- Tuple-at-a-time processing
- Only 1 Join implementation
- Next to no query optimization
- „Ancient“ and extremely popular

DuckDB
- Column-Store
- In-Memory with out-of-memory option
- Vector-at-a-time processing
- Different Join implementations
- Optimizer actually does stuff (join order optimization, eliminate common subqueries,...)
- Relatively new and less popular than SQLite



We already talked about indexes in the first lecture this semester

Reusability of queries and query results

➔ Queries and Subqueries (Views) can be stored and referenced → Nicer queries

➔ The result of views can be stored → Higher performance for frequently used queries and remote data

Exercise Optimizations by the user

Query 1

SELECT count(*) FROM testsequence, authors WHERE testsequence.Sequence_Count < 3 AND testsequence.PDB_ID = authors.IDCODE;

Query 2

SELECT authors.IDCODE,authors.name FROM institute, testsequence, authors WHERE testsequence.Sequence_Count < 3 AND testsequence.PDB_ID = authors.IDCODE AND institute.name = authors.institute;

Which optimizations can a user apply to increase the query performance?

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG