Introduction to Database Systems
Spring 2026, Lecture 9:
Join Algorithms,
Min/Max Summaries

Image source: My Espresso Shop

Intended Learning Outcomes

1. Suggest a database design in the ER model and convert to a relational database schema in a suitable normal form.
2. Write SQL queries, involving multiple relations, compound conditions, grouping, aggregation, and subqueries.
3. Use relational DBMSs from a conventional programming language in a secure manner.
4. Analyze/predict/improve query processing efficiency of the designed database using indices.
5. Reflect upon the evolution of the hardware and storage hierarchy and its impact on data management system design.
6. Discuss the pros and cons of different classes of data systems for modern analytics and data science applications.

Context

Lectures until now
(How to use a database)

Lectures from now
(How a database works)

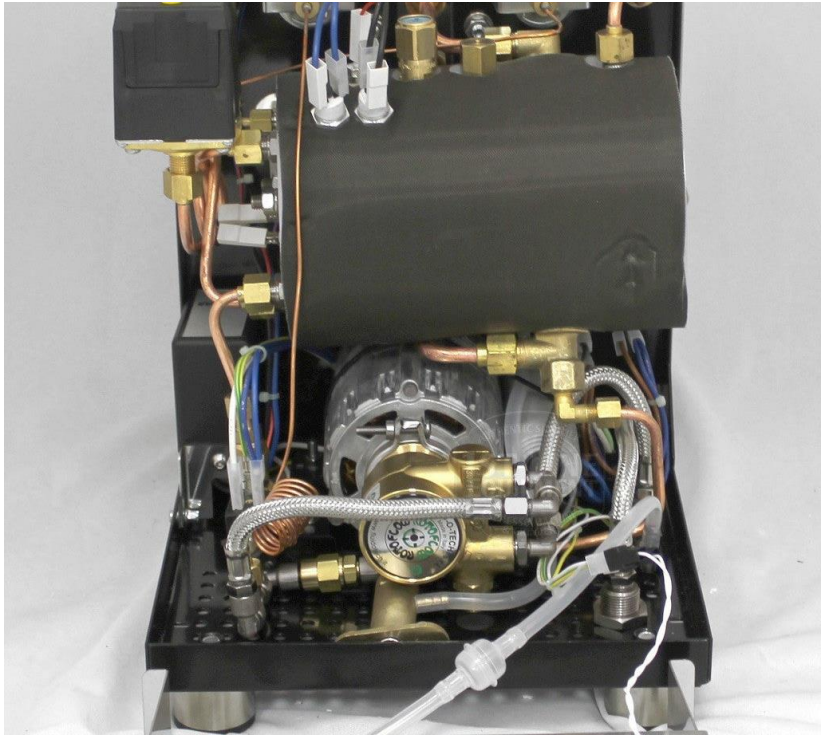


Image sources: My Espresso Shop

Agenda

Architecture of a Database

Join Algorithms

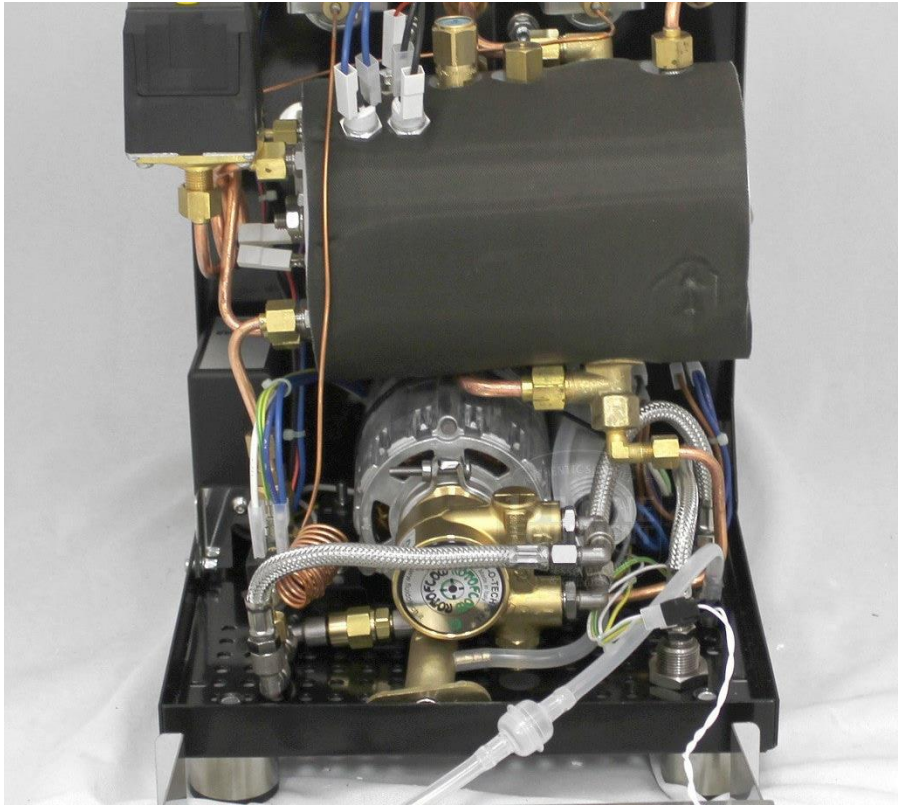
- Nested loop join
- Hash join
- Sort-merge join

Min/Max Summarizes

- Overview
- Pruning Potential

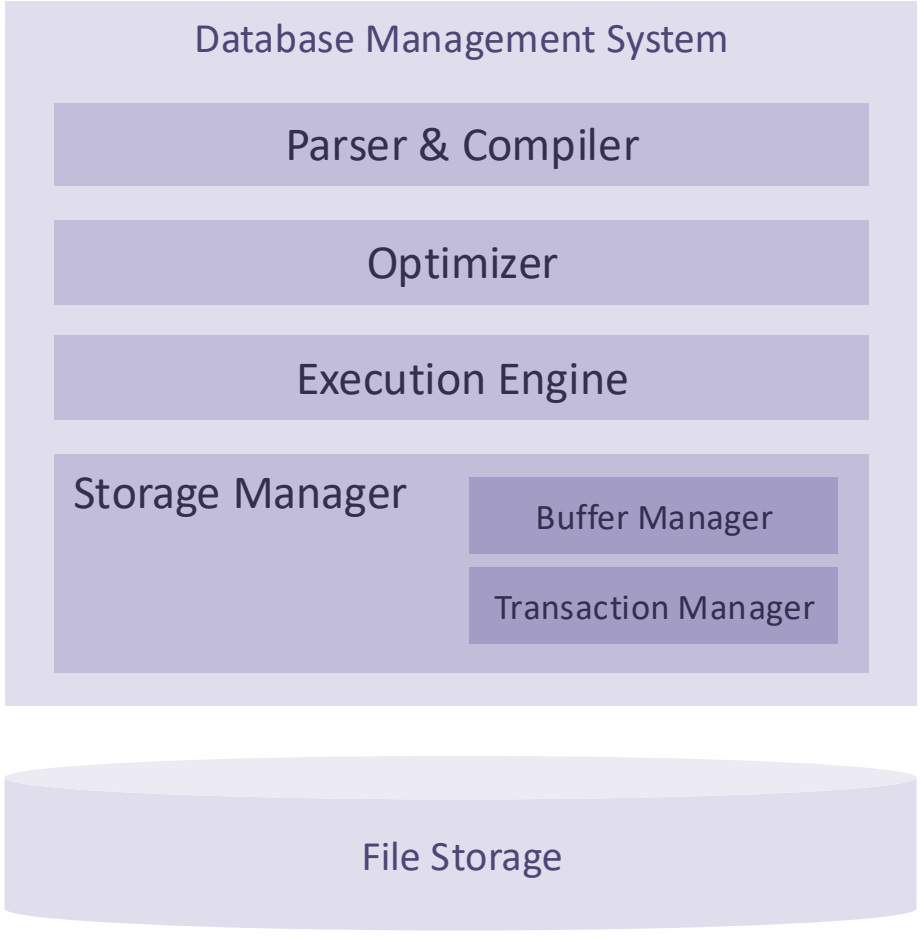
Architecture of a Database

Important components of a database system



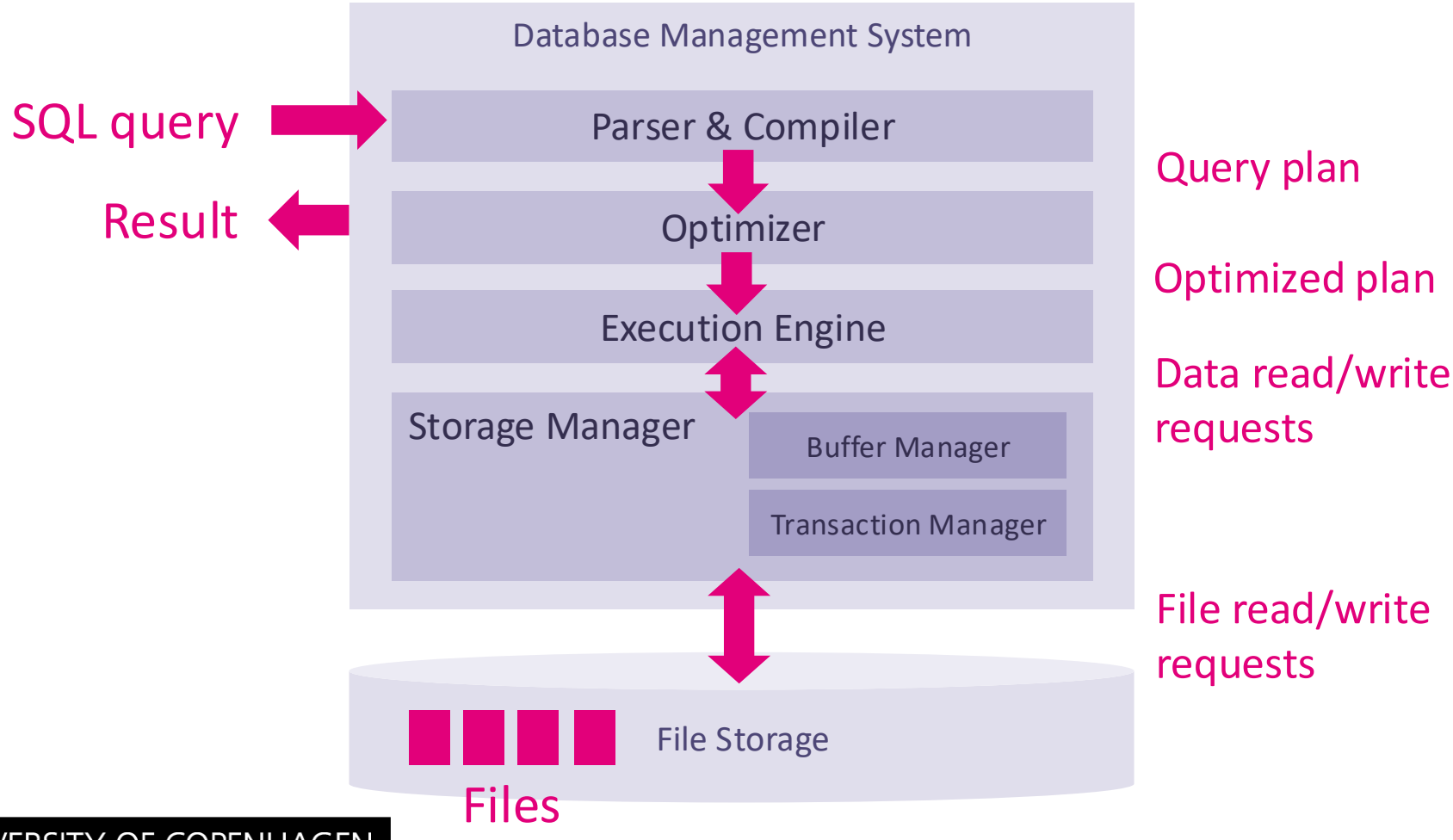
Architecture of a Database*

Important components of a database system

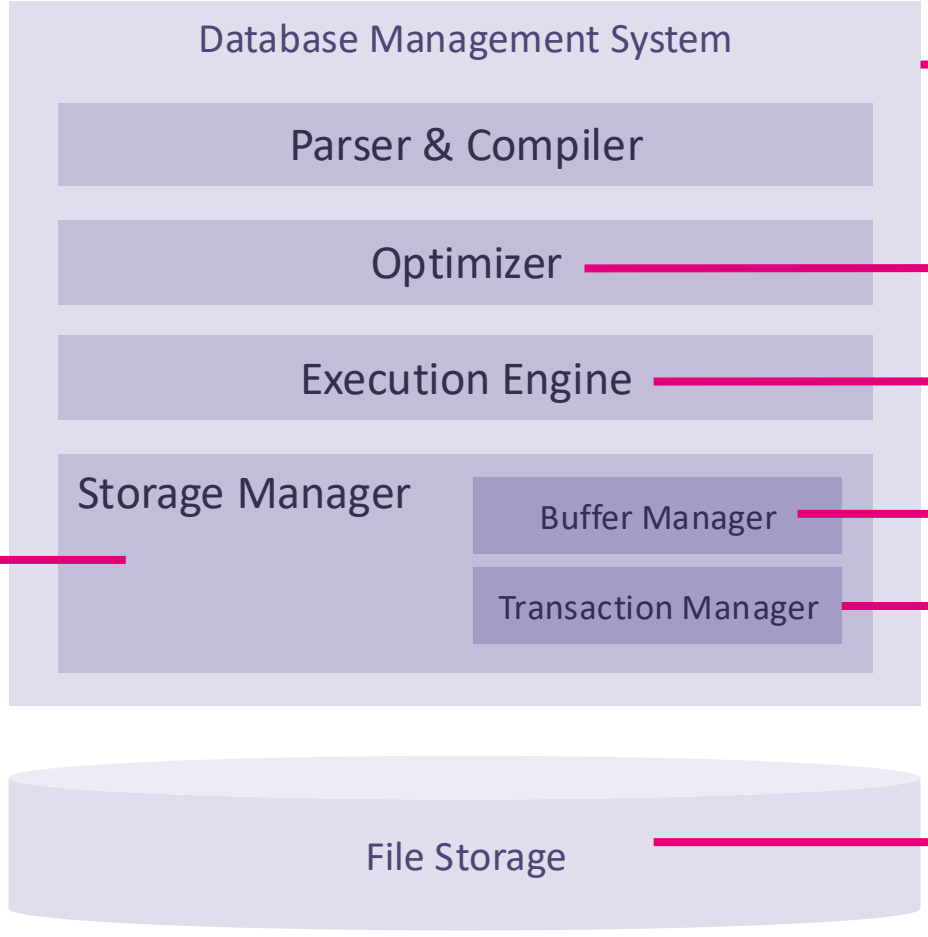


*Only a few important components shown. Many more components exist in practice, varying by database system.

Flow of Information



Components Covered in this Course



Big data systems
(Lecture 12)

Indexes
(Lecture 10)

Join algorithms
(this lecture)

Min/max summaries
(this lecture)

B-trees
(Lecture 10)

Page loading
(Lecture 10)

Transactions, ACID
(Lecture 8)

OLTP vs. OLAP
(Lecture 11)

Agenda

Architecture of a Database

Join Algorithms
- Nested loop join
- Hash join
- Sort-merge join

Min/Max Summarizes
- Overview
- Pruning Potential

Inner Join on Equality

```
select *
from purchase inner join product
using (productname);
```

purchasetime	productname	username	price
2026-02-11 09:55:00	Tea	Martin	100
2026-02-12 10:03:00	Small	Martin	85
2026-02-12 10:05:00	Small	Omar	85
2026-02-12 10:06:00	Large	Omar	100
2026-02-19 09:00:00	Small	Martin	85

Purchase

PurchaseTime	ProductName	UserName
Feb 11, 9:55	Tea	Martin
Feb 12, 10:03	Small	Martin
Feb 12, 10:05	Small	Omar
Feb 12, 10:06	Large	Omar
Feb 19, 9:00	Small	Martin

Product

ProductName	Price
Tea	100
Small	85
Large	100
Fancy	NULL

How to implement an equality join in Java or Python?

We will look at 3 different options.

Java Records

Records in Java holding purchase tuples, product tuples, and results

```
public record Purchase(
    String time, String productName, String userName) {}

public record Product(
    String productName, int price) {}

public record Result(
    Purchase purchase, Product product) {}
```

Python Classes

Python classes holding purchase tuples, product tuples, and results

```
@dataclass
class Purchase:
    time: str
    product_name: str
    user_name: str

@dataclass
class Product:
    product_name: str
    price: int

@dataclass
class Result:
    purchase: Purchase
    product: Product
```

Option 1: Nested Loop Join

Two nested loops: an outer loop & an inner loop.

- For every purchase row, iterate over every product row and check if the product names match

Purchase (PurchaseTime, ProductName, UserName)

PurchaseTime	ProductName	UserName
Feb 11, 9:55	Tea	Martin
Feb 12, 10:03	Small	Martin
Feb 12, 10:05	Small	Omar
Feb 12, 10:06	Large	Omar
Feb 19, 9:00	Small	Martin

Product (ProductName, Price)

ProductName	Price
a	100
all	85
rge	100
ncy	NULL

5 * 4 comparisons → O(N * M)
~ quadratic runtime

Nested Loop Join in Java

```
// nested loop join
public List<Result> joinNestedLoops(List<Purchase> purchases,
                                    List<Product> products) {
    List<Result> result = new ArrayList<>();

    for (Purchase purchase : purchases) {
        for (Product product : products) {
            if (purchase.productName().equals(product.productName())) {
                result.add(new Result(purchase, product));
            }
        }
    }

    return result;
}
```

Nested Loop Join in Python

```
def join_nested_loops(self, purchases, products):
    """Nested loop join algorithm"""
    result = []

    for purchase in purchases:
        for product in products:
            if purchase.product_name == product.product_name:
                result.append(Result(purchase, product))

    return result
```

Option 2: Hash Join

A hash table and a single loop.

- Build a hash table for product rows using product names as keys. For each purchase row, look up the matching product row in the hash table.

Purchase (PurchaseTime, ProductName, UserName)

PurchaseTime	ProductName	UserName
Feb 11, 9:55	Tea	Martin
Feb 12, 10:03	Small	Martin
Feb 12, 10:05	Small	Omar
Feb 12, 10:06	Large	Omar
Feb 19, 9:00	Small	Martin

Product Hash Table

Tea	100
Small	85
Large	100
Fancy	NULL

4 hash table inserts & 5 hash look ups → O(M + N)
~ linear runtime

Hash Join in Java

```java
// hash join
public List<Result> joinHash(List<Purchase> purchases,
                             List<Product> products) {

    List<Result> result = new ArrayList<>();

    Map<String, Product> productMap = new HashMap<>();
    for (Product product : products) {
        productMap.put(product.productName(), product);
    }

    for (Purchase purchase : purchases) {
        Product product = productMap.get(purchase.productName());
        if (product != null) {
            result.add(new Result(purchase, product));
        }
    }

    return result;
}
```

Hash Join in Python

```
def join_hash(self, purchases, products):
    """Hash join algorithm"""

    product_dict = {product.product_name: product for product in products}

    result = []
    for purchase in purchases:
        product = product_dict[purchase.product_name]
        if product is not None:
            result.append(Result(purchase, product))

    return result
```

Option 3: Sort-Merge Join

Sort both tables, iterate over both together.

- Sort purchase rows and product rows by product name. Scan through both sorted lists, comparing product names.

Purchase (PurchaseTime, ProductName, UserName)

PurchaseTime	ProductName	UserName
Feb 12, 10:06	Large	Omar
Feb 12, 10:03	Small	Martin
Feb 12, 10:05	Small	Omar
Feb 19, 9:00	Small	Martin
Feb 11, 9:55	Tea	Martin

Product (ProductName, Price)

ProductName	Price
Fancy	NULL
Large	100
Small	85
Tea	100

2 table sorts & 2 table scans → O(N log N + M log M)

~ log-linear runtime (sort dominates scan)

Sort-Merge Join in Java

```
// sort-merge join
public List<Result> joinSortMerge(List<Purchase> purchases,
                                  List<Product> products) {
    List<Result> result = new ArrayList<>();

    purchases.sort(Comparator.comparing(Purchase::productName));
    products.sort(Comparator.comparing(Product::productName));

    int i = 0, j = 0;
    while (i < purchases.size() && j < products.size()) {
        Purchase purchase = purchases.get(i);
        Product product = products.get(j);
        int compare = purchase.productName().compareTo(product.productName());
        if (compare == 0) {
            result.add(new Result(purchase, product));
            i++; // note we only advance i, not j, because the next purchase
                 // may match the same product
        } else if (compare < 0) {
            i++;
        } else {
            j++;
        }
    }
    return result;
}
```

Sort-Merge Join in Python

```
def join_sort_merge(self, purchases, products):
    """Sort-merge join algorithm"""
    # Sort both lists by product name
    purchases_sorted = sorted(purchases, key=lambda p: p.product_name)
    products_sorted = sorted(products, key=lambda p: p.product_name)

    result = []
    i, j = 0, 0
    while i < len(purchases_sorted) and j < len(products_sorted):
        purchase = purchases_sorted[i]
        product = products_sorted[j]
        if purchase.product_name == product.product_name:
            result.append(Result(purchase, product))
            i += 1  # note we only advance i, not j, because
                    # because the next purchase may match the same product
        elif purchase.product_name < product.product_name:
            i += 1
        else:
            j += 1
    return result
```

Benchmarking the Java Implementation

```
// create benchmark data
int productCount  = 1000;
int purchaseCount = 10_000_000;
List<Purchase> purchases = createPurchases(purchaseCount, productCount);
List<Product> products   = createProducts(productCount);

// benchmark three join algorithms
benchmark("nested loop join", lecture8::joinNestedLoops, purchases, products);
benchmark("hash join",        lecture8::joinHash,        purchases, products);
benchmark("sort-merge join",  lecture8::joinSortMerge,   purchases, products);
```

Result:

```
nested loop join, result size: 10000000, time: 33010 ms
hash join,        result size: 10000000, time:   514 ms
sort-merge join,  result size: 10000000, time:  5274 ms
```

Big differences!

Benchmarking the Python Implementation

```
# create benchmark data
product_count = 1000
purchase_count = 10_000_000
purchases = self.create_purchases(purchase_count, product_count)
products = self.create_products(product_count)

# benchmark three join algorithms
self.benchmark("nested loops join", self.join_nested_loops, purchases, products)
self.benchmark("hash join", self.join_hash, purchases, products)
self.benchmark("sort-merge join", self.join_sort_merge, purchases, products)
```

Result:

```
nested loop join, result size: 10000000, time: 176779 ms
hash join,        result size: 10000000, time:   6859 ms
sort-merge join,  result size: 10000000, time:  16554 ms
```

Similar big differences!

Benchmarking DuckDB

Export to CSV and import into DuckDB

```
~$ ls -lh *.csv
-rwxrwxrwx 1 mhent mhent  17K Oct  9 14:17 products.csv
-rwxrwxrwx 1 mhent mhent 374M Oct  9 14:17 purchases.csv
```

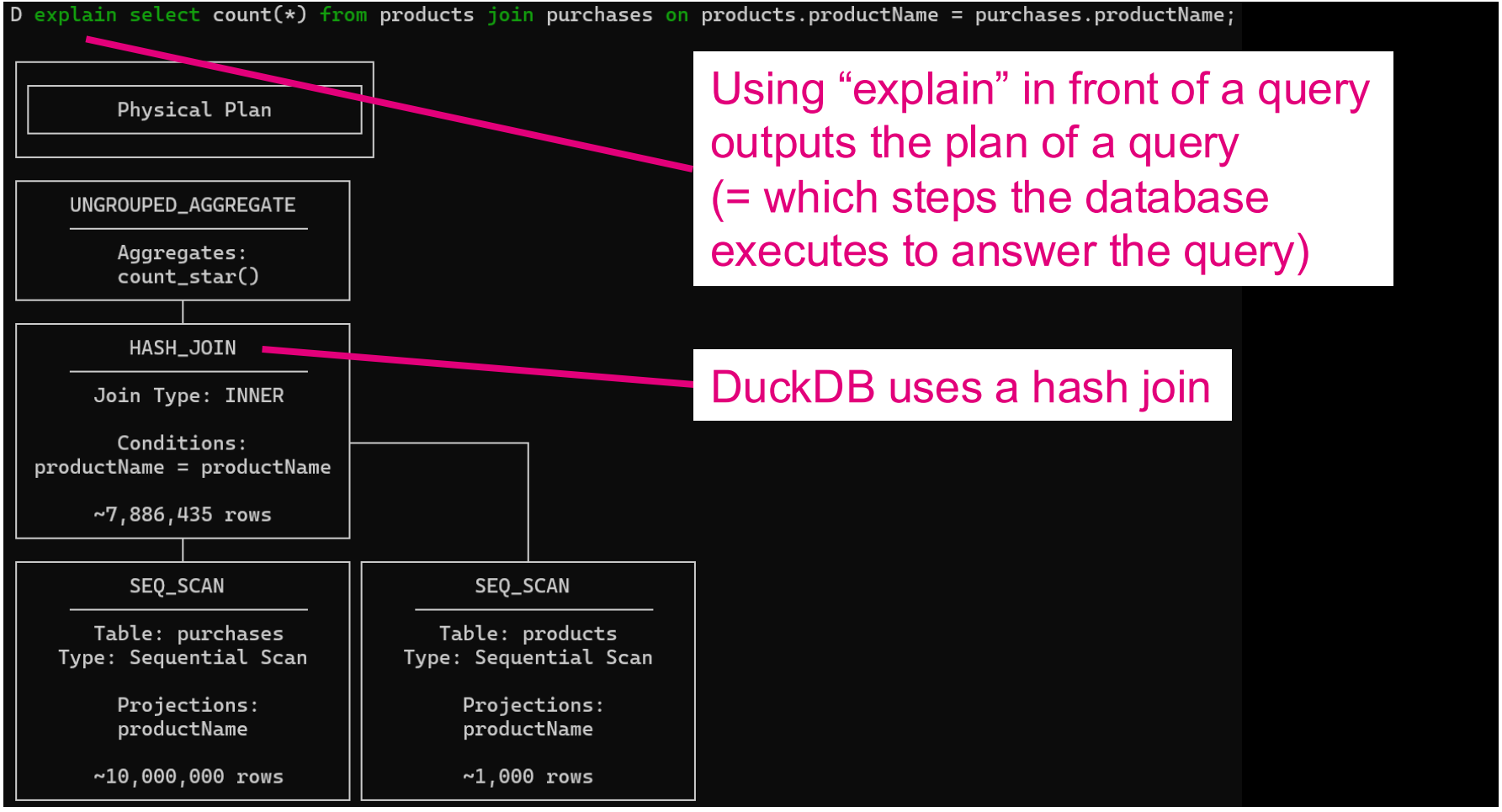
10 million purchases =
374MB of data

Run the join in DuckDB

```
D select count(*) from products join purchases on products.productName = purchases.productName;
┌──────────────┐
│ count_star() │
│    int64     │
├──────────────┤
│   10000000   │
│ (10.00 million) │
└──────────────┘
Run Time (s): real 0.079 user 0.258583 sys 0.035263
```

79 ms in DuckDB vs.
514 ms in the Java hash join implementation
(DuckDB is multi-threaded and highly optimized)

DuckDB Query Plan

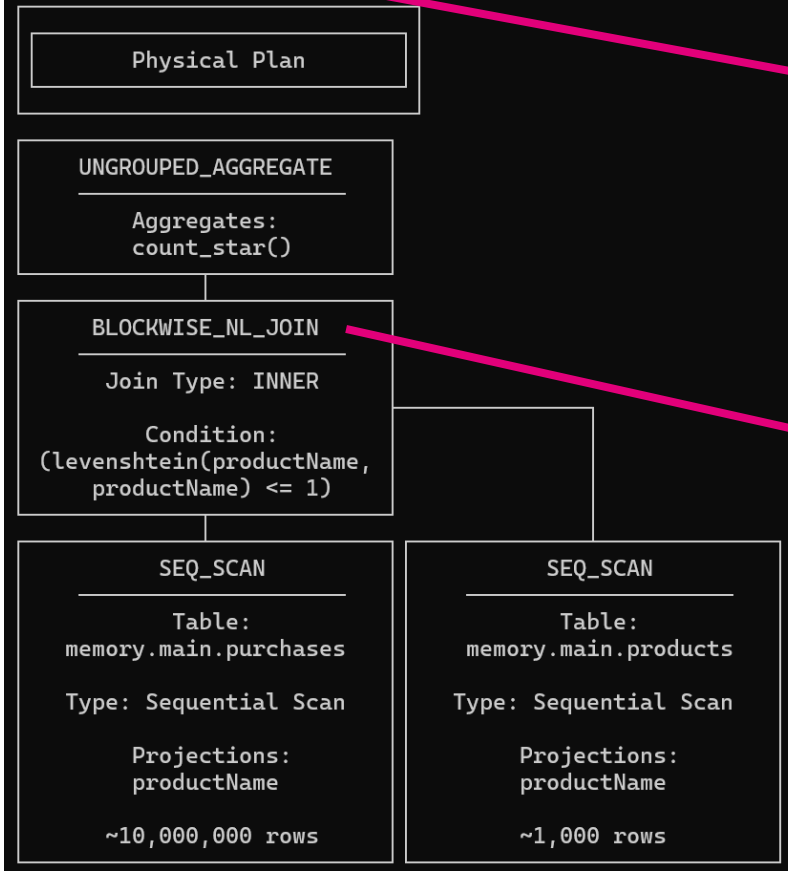


Using “explain” in front of a query outputs the plan of a query (= which steps the database executes to answer the query)

DuckDB uses a hash join

DuckDB Query Plan for a Non-Equality Join

```
memory D explain select count(*) from products join purchases
         on levenshtein(products.productName, purchases.productName) <= 1;
```



Levenshtein distance from last week's lecture

Nested Loop Join
(takes forever if you run it on this dataset)

Notes on Join Algorithms

Nested Loop Join

- Must be two loops. If inner loop is replaced by an index look-up, it's not a nested loop join anymore.
- Relevant for joins without equality predicates. Specialized other join algorithms exist different use cases.

Sort-Merge Join

- Useful when data is already sorted and sorting step can be skipped.

Hash Join

- Most common in practice! Always index the smaller relation in the hash table.

Agenda

Architecture of a Database

Join Algorithms
- Nested loop join
- Hash join
- Sort-merge join

Min/Max Summarizes
- Overview
- Pruning Potential

Data in Files

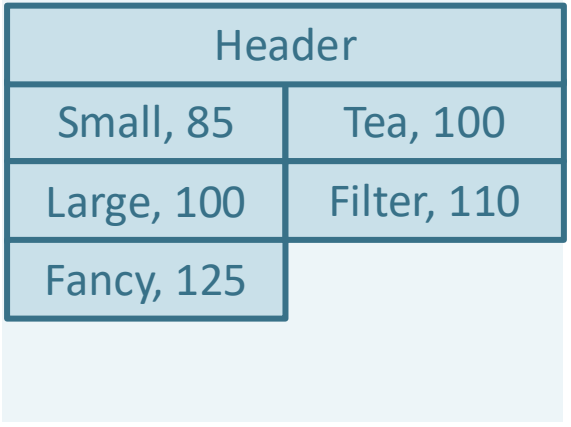
Store data in files by tuples, one tuple after the next

- Header contains metadata like number of tuples, start byte of first tuple, checksums, and more
- Tuples can be of different lengths (for simplicity all same length = width below)

Product (ProductName, Price)

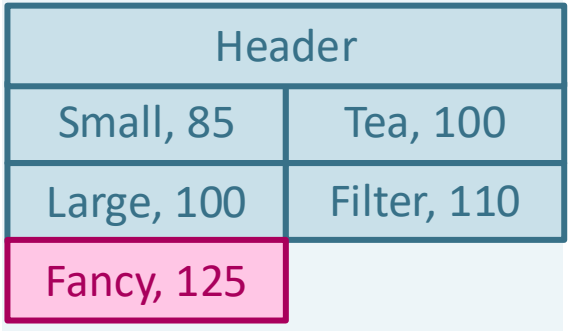
ProductName	Price
Small	85
Tea	100
Large	100
Filter	110
Fancy	125

Product.data



Query Processing Example

Product.data



Execution engine executes

```
select * from product
where productname = 'Fancy';
```

Steps:

- Read header of Product.data
- Jump to first tuple
- Read tuple and compare name ‘Small’ to ‘Fancy’
  if match → add tuple to result set
- Jump to next tuple
- …repeat…

Repeats for all tuples, in total 5 tuple reads and 5 comparisons

Min/Max Summaries

That was a “full table scan”

→ Can lead to bad performance if table gets large

→ How to optimize full table scans? One technique that databases implement are:

Min/Max Summaries

also called: Min/max indexes
Zone maps
Block range indexes
Small materialized aggregates

See also: [Building a Data Management System for the Cloud: Lessons Learned and Future Directions]

File Structure

When data gets big, the database may partition the data

- Partition = row group = a group of tuples within a file
- Split data into multiple files

10 tuples now for a better example

Product.data

Header	
Small, 85	Tea, 100
Large, 100	Filter, 110
Fancy, 125	Espresso, 35
Cortado, 35	Mocha, 45
Jasmin, 30	Cola, 25

partition into groups of rows

Product.data

Header	
Small, 85	Tea, 100
Large, 100	Filter, 110
Fancy, 125	Espresso, 35
Cortado, 35	Mocha, 45
Jasmin, 30	Cola, 25

Row group 1

Row group 2

Row group 3

Collecting Statistics

Partitioning happens during insertion of data

→ Possibility to collect statistics: min/max summaries

→ Stored e.g., in footer of the file (Apache Parquet file format)

Product.data

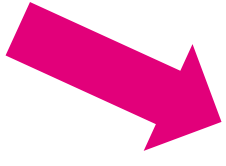
Header	
Small, 85	Tea, 100
Large, 100	Filter, 110
Fancy, 125	Espresso, 35
Cortado, 35	Mocha, 45
Jasmin, 30	Cola, 25

Min/max summaries per column for each row group

Name: Filter – Tea
Price: 85 – 110

Name: Cortado – Mocha
Price: 35 – 125

Name: Cola – Jasmin
Price: 25 – 30



Product.data

Header	
Small, 85	Tea, 100
Large, 100	Filter, 110
Fancy, 125	Espresso, 35
Cortado, 35	Mocha, 45
Jasmin, 30	Cola, 25
Footer	

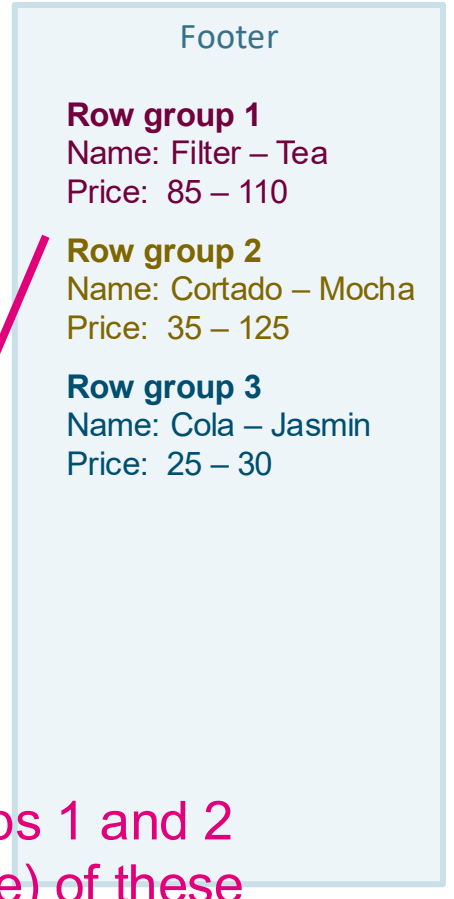
Using Statistics During Compilation

Query optimizer decides which row groups to skip based on query predicate

- This is called “pruning”
- Usually happens at compile time, before executing the query

```
select * from product
where productname = 'Cola';
```

Predicate is productname = ‘Cola’



Optimizer skips row groups 1 and 2 because min(productname) of these row groups > ‘Cola’ (alphabetically)

Query Plan after Optimization

```
select * from product
where productname = 'Cola';
```

Query plan before optimization

- Read all tuples of file Product.data
- If productname = 'Cola' → add to result

  = 10 tuple reads + 10 comparisons

Query plan after optimization

- Read all tuples from **row group 3 only**
- If productname = 'Cola' → add to result

  = 2 tuple reads + 2 comparisons

Footer

Row group 1
Name: Filter – Tea
Price: 85 – 110

Row group 2
Name: Cortado – Mocha
Price: 35 – 125

Row group 3
Name: Cola – Jasmin
Price: 25 – 30

Live Exercise

Which row groups to read?

```
select * from product
where price >= 100;
```

Result: Row groups 1 and 2
Row group 3 pruned because max(price) < 100

```
select * from product
where productname = 'Mocha'
and price < 60;
```

Result: Row group 2
Row group 3 pruned because max(name) < 'Mocha'
Row group 1 pruned because min(price) >= 60

```
select min(productName) from product;
```

Result: No row groups need to be read
Query can be answered using min/max statistics alone

Footer

Row group 1
Name: Filter – Tea
Price: 85 – 110

Row group 2
Name: Cortado – Mocha
Price: 35 – 125

Row group 3
Name: Cola – Jasmin
Price: 25 – 30

Min/Max Summaries of Files

Also works with files instead of (or in combination with) row groups

→ Min/max summaries per file (or per file *and* row group)

→ Min/max metadata may be stored outside of files

Product1.data

Header	
Small, 85	Tea, 100
Large, 100	Filter, 110

Product2.data

Header	
Fancy, 125	Espresso, 35
Cortado, 35	Mocha, 45

Product3.data

Header	
Jasmin, 30	Cola, 25

Minmax.metadata

Product1.data
Name: Filter – Tea
Price: 85 – 110

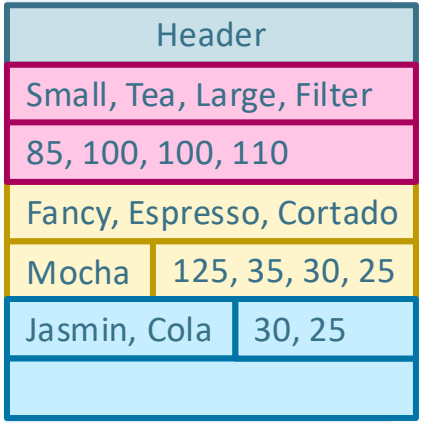
Product2.data
Name: Cortado – Mocha
Price: 35 – 125

Product3.data
Name: Cola – Jasmin
Price: 25 – 30

Row Groups vs. Files Examples

DuckDB

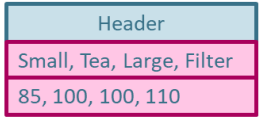
Product.duckdb



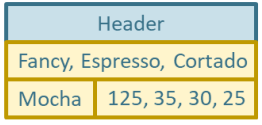
1 file split into multiple row groups

snowflake

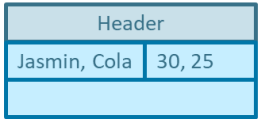
Product1.data



Product2.data



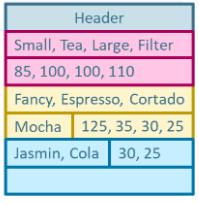
Product3.data



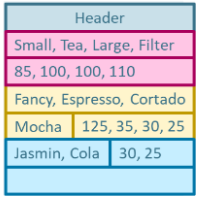
Multiple files each with one row group

Parquet

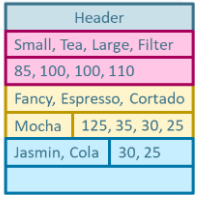
Product1.parquet



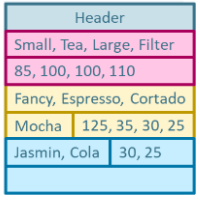
Product2.parquet



Product3.parquet



Product4.parquet



Multiple files each split into multiple row groups

Agenda

Architecture of a Database

Join Algorithms
- Nested loop join
- Hash join
- Sort-merge join

Min/Max Summarizes
- Overview
- Pruning Potential

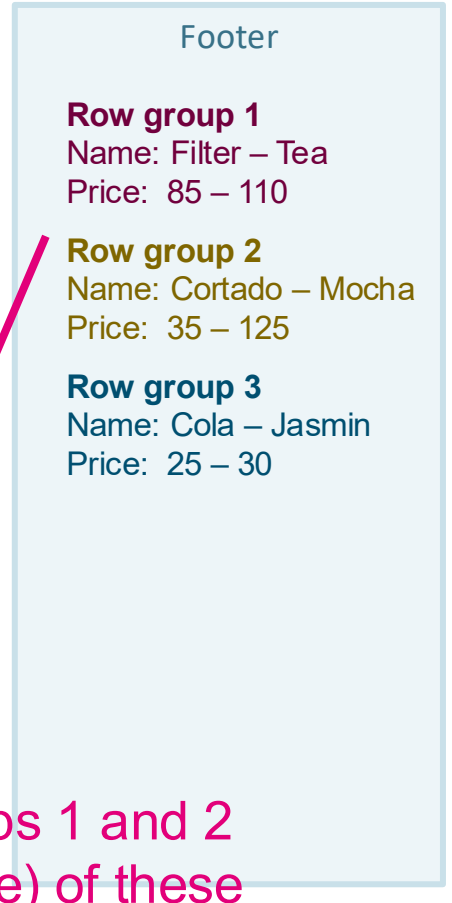
Pruning Repeated

Query optimizer decides which row groups to skip based on query predicate

- This is called “pruning”
- Usually happens at compile time, before executing the query

```
select * from product
where productname = 'Cola';
```

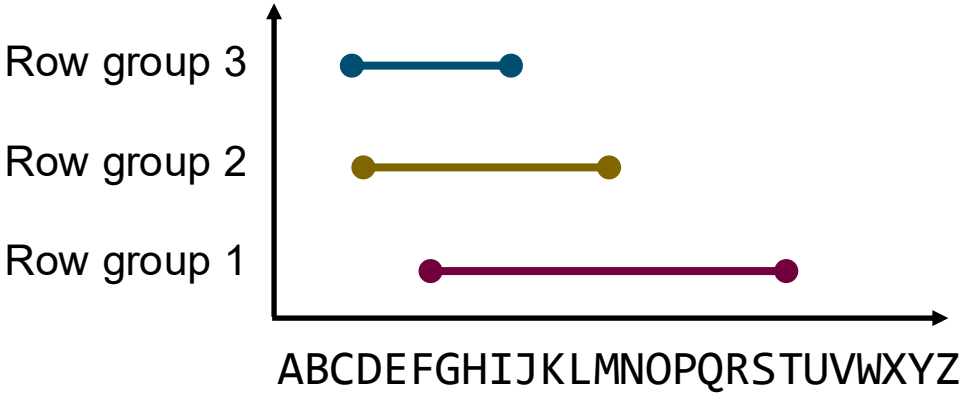
Predicate is productname = ‘Cola’



Optimizer skips row groups 1 and 2 because min(productname) of these row groups > ‘Cola’

Visualization of Min/Max Summaries

Each line represents the min and max values for column ‘name’ per row group



Footer

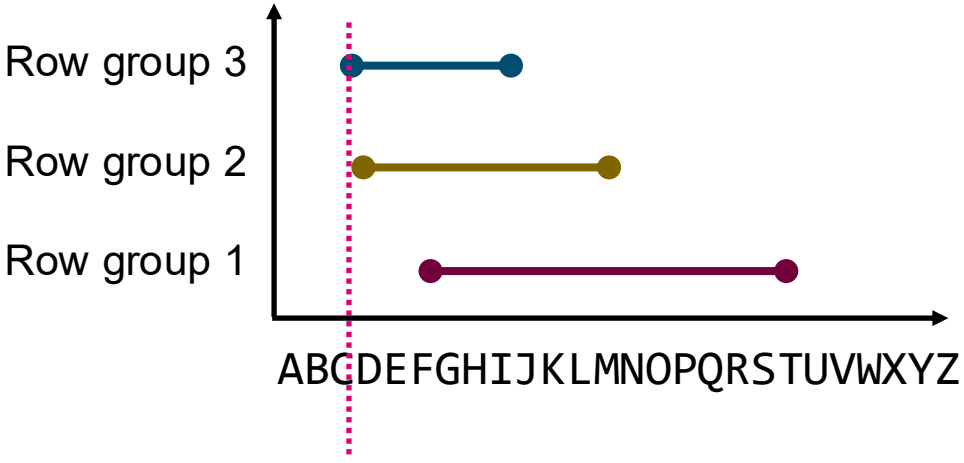
Row group 1
Name: Filter – Tea
Price: 85 – 110

Row group 2
Name: Cortado – Mocha
Price: 35 – 125

Row group 3
Name: Cola – Jasmin
Price: 25 – 30

Visualization of Query Predicate ‘Cola’

Query predicate is a vertical line in the chart: if it cuts min/max → cannot prune



Predicate
productname = ‘Cola’
→ Cannot prune row group 3
→ But prune row groups 1 and 2

Footer

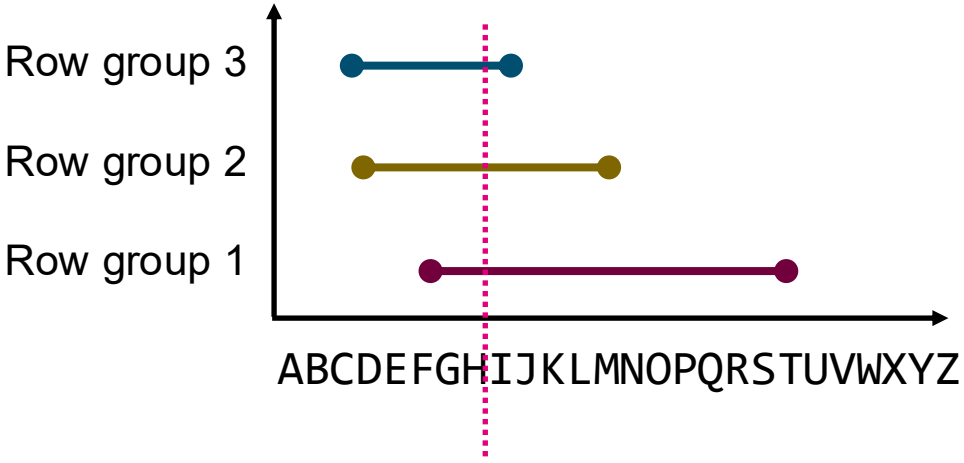
Row group 1
Name: Filter – Tea
Price: 85 – 110

Row group 2
Name: Cortado – Mocha
Price: 35 – 125

Row group 3
Name: Cola – Jasmin
Price: 25 – 30

Visualization of Query Predicate ‘Hot Tea’

Query predicate is a vertical line in the chart: if it cuts min/max → cannot prune



Predicate
productname = ‘Hot Tea’
→ No pruning

Footer

Row group 1
Name: Filter – Tea
Price: 85 – 110

Row group 2
Name: Cortado – Mocha
Price: 35 – 125

Row group 3
Name: Cola – Jasmin
Price: 25 – 30

Sorting the Table by ProductName

Sorted by ProductName:

Product.data

Header	
Cola, 25	Cortado, 35
Espresso, 35	Fancy, 125
Filter, 110	Jasmin, 30
Large, 100	Mocha, 45
Small, 85	Tea, 100

Min/max summaries per column for each row group

Name: Cola – Fancy
Price: 25 – 125

Name: Filter – Mocha
Price: 30 – 110

Name: Small – Tea
Price: 85 – 100

Different summaries because now data is sorted by product name

Footer

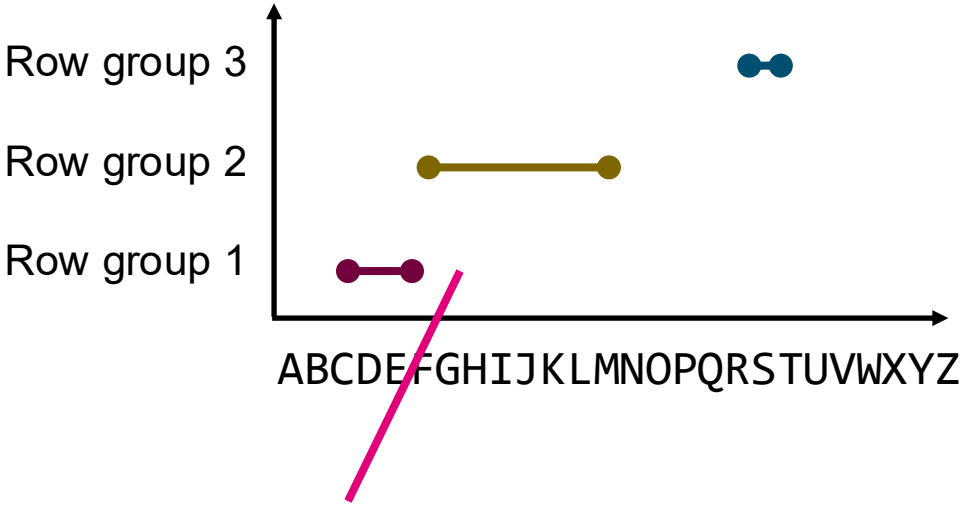
Row group 1
Name: Cola – Fancy
Price: 25 – 125

Row group 2
Name: Filter – Mocha
Price: 30 – 110

Row group 3
Name: Small – Tea
Price: 85 – 100

Visualization of Min/Max Summaries

Sorted by ProductName:



No overlapping min/max summaries
→ Good pruning potential for any predicate on productname

Footer

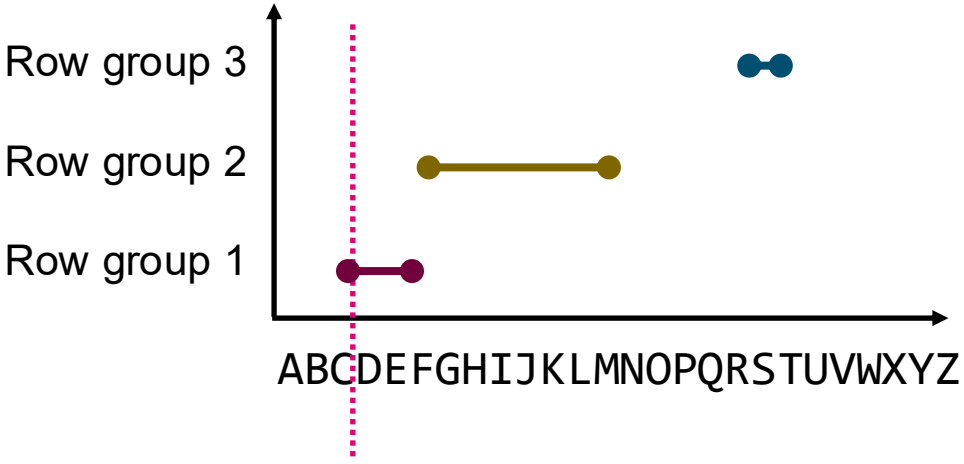
Row group 1
Name: Cola – Fancy
Price: 25 – 125

Row group 2
Name: Filter – Mocha
Price: 30 – 110

Row group 3
Name: Small – Tea
Price: 85 – 100

Visualization of Query Predicate ‘Cola’

Perfect pruning for ‘Cola’



Predicate
productname = ‘Cola’
→ Prune row groups 2 and 3

Footer

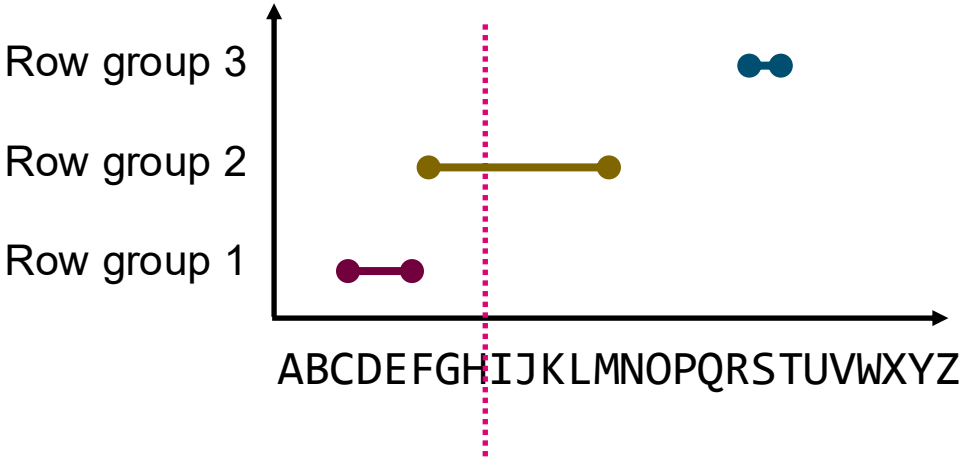
Row group 1
Name: Filter – Tea
Price: 85 – 110

Row group 2
Name: Cortado – Mocha
Price: 35 – 125

Row group 3
Name: Cola – Jasmin
Price: 25 – 30

Visualization of Query Predicate ‘Hot Tea’

Perfect pruning for ‘Hot Tea’



Predicate
productname = ‘Hot Tea’
→ Prune row groups 1 and 3

Footer

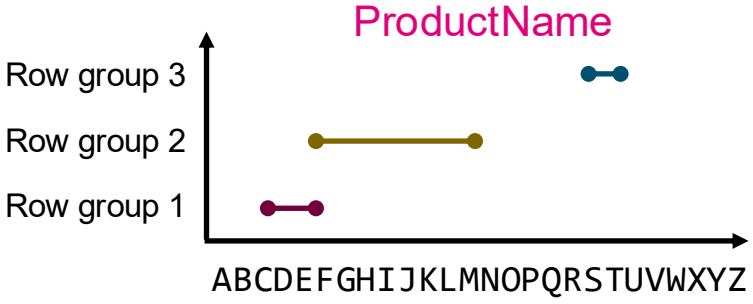
Row group 1
Name: Filter – Tea
Price: 85 – 110

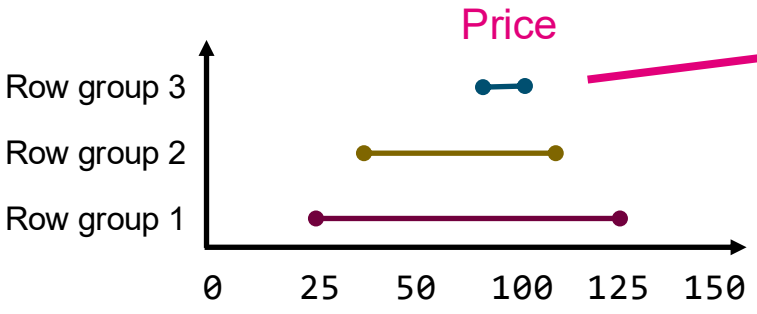
Row group 2
Name: Cortado – Mocha
Price: 35 – 125

Row group 3
Name: Cola – Jasmin
Price: 25 – 30

What about Column ‘Price’?

Plots of min/max summaries per column





Overlapping summaries → lower pruning potential when filtering by ‘price’

Footer

Row group 1
Name: Cola – Fancy
Price: 25 – 125

Row group 2
Name: Filter – Mocha
Price: 30 – 110

Row group 3
Name: Small – Tea
Price: 85 – 100

Pruning Potential of Min/Max Summaries

Pruning potential of min/max summaries depends on data ordering:

- Sorted or mostly sorted data → high pruning potential
- Unsorted data → low pruning potential

Understanding query performance:

→ If your queries primarily filter on one column, create a table that is ordered by that column

→ If queries filter on different columns, consider adding an index on the other columns (next lecture)

Summary of Min/Max Summaries

During data insertion

- Partition data into row groups and/or files
- Collect min/max summaries per partition

During query compilation

- Prune partitions based on min/max summaries and query predicates → less data to read → faster execution
- Pruning also possible during execution (not covered today)
- Pruning potential depends on data ordering

Takeaways

Joins can be implemented differently

→ Big impact on performance

→ Why care? If a query plan uses a slow join implementation, try to rewrite the query to make it faster.

Min/max summaries avoid reading all data

→ Big impact on performance

→ Why care? Sorting / clustering of data can make your data jobs run faster.