Declarative Languages and Relational Algebra

CSE 4/562: Database Systems | Lecture 2

DB. Sys.: T.C.B.: 2.1-2.4, 5.1

SQL vs Python

What is different about these code snippets?

```sql
SELECT R.A, SUM(S.B)
FROM R, S
WHERE R.B = S.B
```

```python
for r in R:
  lookup[r.B] += [r.A]
for s in S:
  for a in lookup[s.B]:
    result[a] += s.C
```

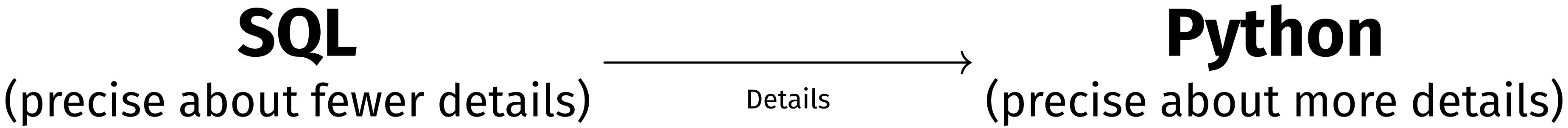
Declarative Languages

```
SELECT R.A, SUM(S.B)
FROM R, S
WHERE R.B = S.B
```

```
for r in R:
  lookup[r.B] += [r.A]
for s in S:
  for a in lookup[s.B]:
    result[a] += s.C
```

- The python code preprocesses R and scans S (and not visa versa)
- The python code explicitly creates a lookup table on R.
- The python code explicitly uses a dictionary for the lookup table.
- The python code explicitly encodes the result as a dictionary.
- SQL only states sources, constraints, and output format.

SQL vs Python



SQL

(precise about fewer details)

Details

Python

(precise about more details)

SQL is a **Declarative Language**

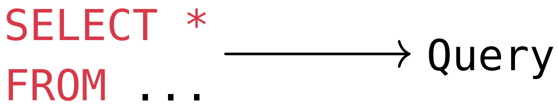
Existing Systems

Postgresql

```
SELECT *
FROM ...
```

Existing Systems

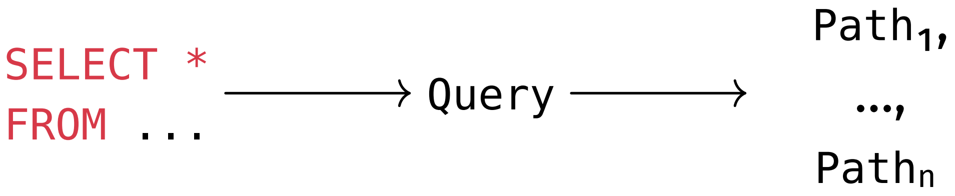
Postgresql



```
SELECT *
FROM ...
```

Existing Systems

Postgresql

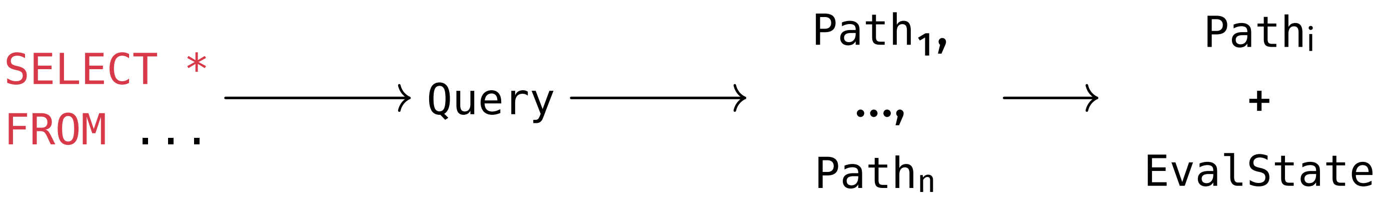


```
SELECT *
FROM ...
```

→ Query → Path_1, ..., Path_n

Existing Systems

Postgresql



```
SELECT *
FROM ...
```

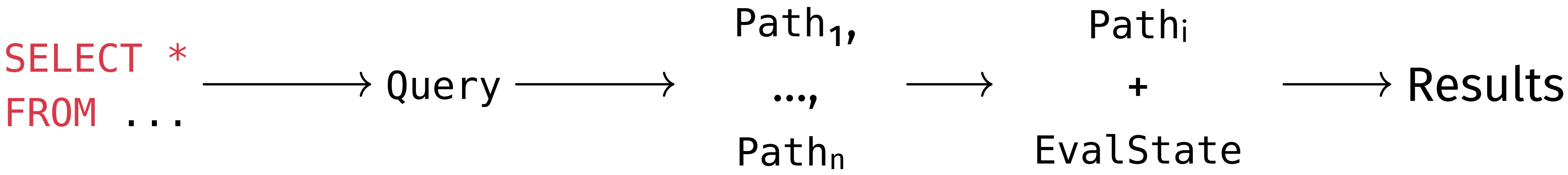
Query

Path$_1$, ..., Path$_n$

Path$_i$ + EvalState

Existing Systems

Postgresql

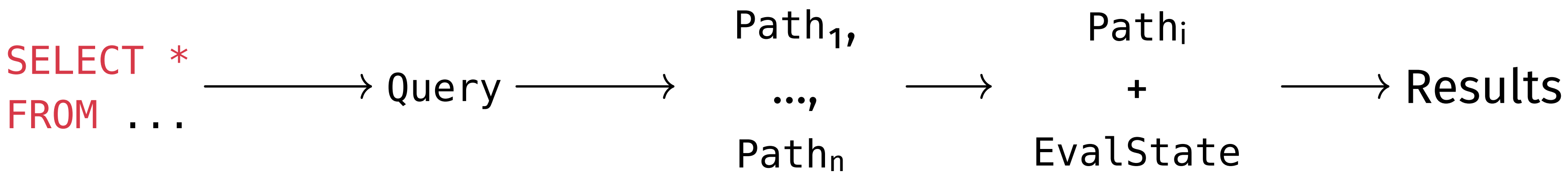


```
SELECT *
FROM ...
```

→ Query → Path_1, ..., Path_n → Path_i + EvalState → Results

Existing Systems

Postgresql



```
SELECT *
FROM ...
```

Query

Path$_1$, ..., Path$_n$

Path$_i$ + EvalState

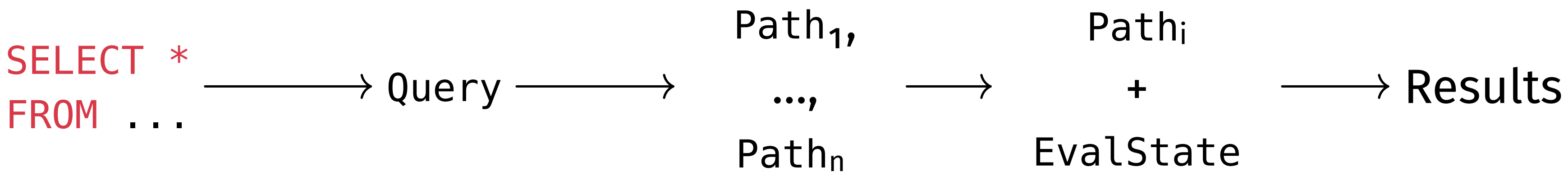
Results

Apache Spark

```
SELECT *
FROM ...
```

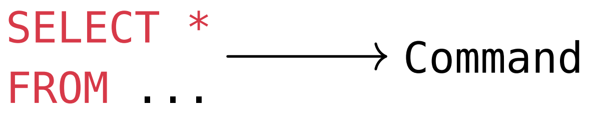
Existing Systems

Postgresql



SELECT * FROM ... → Query → Path_1, ..., Path_n → Path_i + EvalState → Results

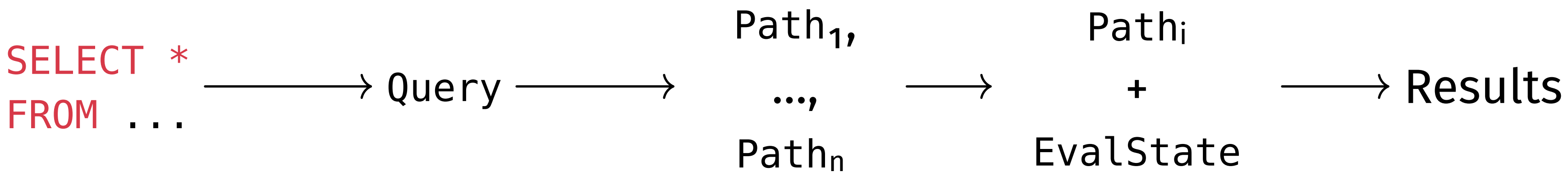
Apache Spark



SELECT * FROM ... → Command

Existing Systems

Postgresql



```
SELECT *
FROM ...
```

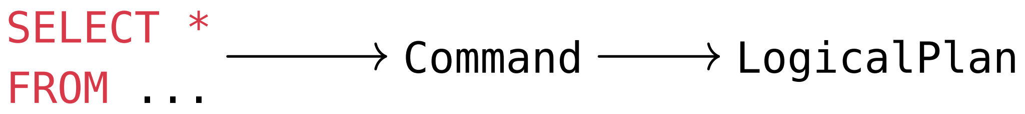
Query

$Path_1$, ..., $Path_n$

$Path_i$ + EvalState

Results

Apache Spark



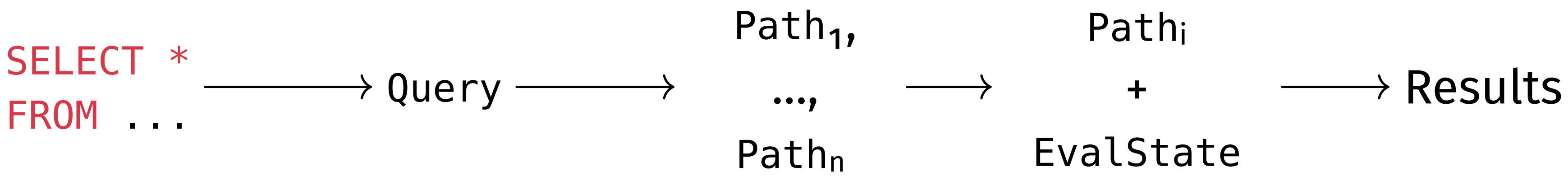
```
SELECT *
FROM ...
```

Command

LogicalPlan

Existing Systems

Postgresql



SELECT *
FROM ...

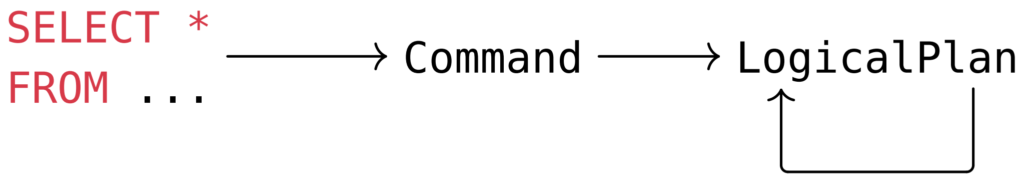
Query

Path$_1$,
...,
Path$_n$

Path$_i$
+
EvalState

Results

Apache Spark



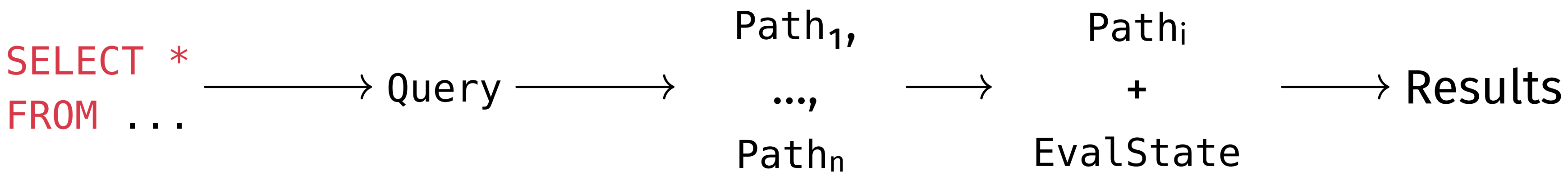
SELECT *
FROM ...

Command

LogicalPlan

Existing Systems

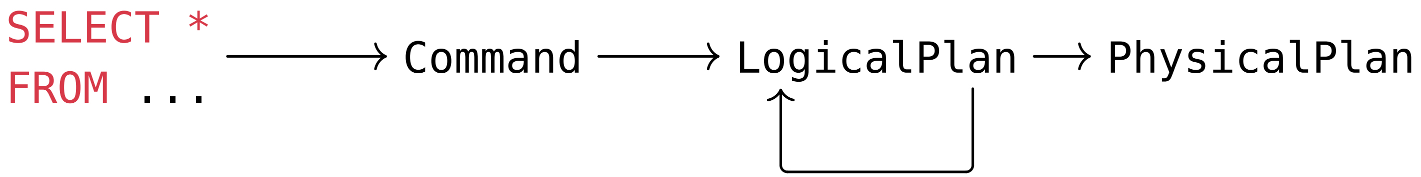
Postgresql



```
SELECT *
FROM ...
```

→ Query → Path_1, …, Path_n → Path_i + EvalState → Results

Apache Spark

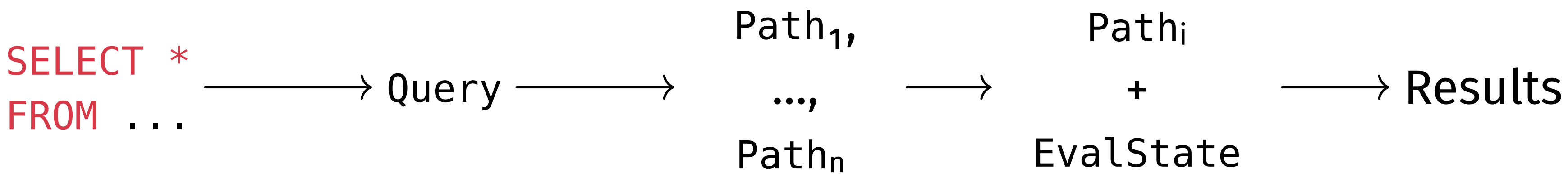


```
SELECT *
FROM ...
```

→ Command → LogicalPlan → PhysicalPlan

Existing Systems

Postgresql



SELECT * FROM ...

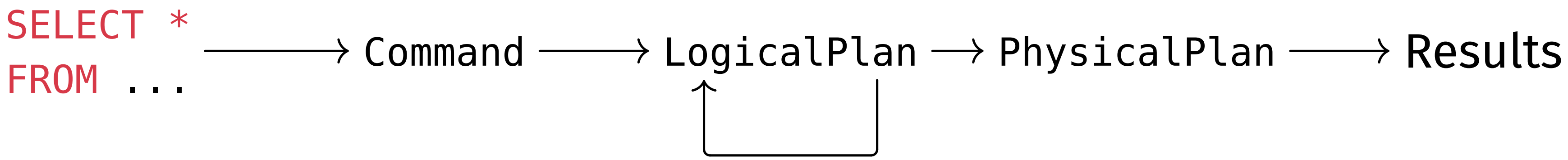
Query

Path$_1$, ..., Path$_n$

Path$_i$ + EvalState

Results

Apache Spark



SELECT * FROM ...

Command

LogicalPlan

PhysicalPlan

Results

Existing Systems: Summary

1. Parse the query
2. Make a Dumb™ plan
3. Come up with better plan ideas and pick one
4. Fill in the specific algorithms and data structures

How do we express a plan?

SQL: Bad for plans?

SQL is Redundant

```
SELECT A, COUNT(DISTINCT B)
FROM R
WHERE C > 5
GROUP BY A
HAVING SUM(C) < 20
```

SQL: Bad for plans?

SQL is Redundant

```
SELECT A, COUNT(DISTINCT B)
FROM R
WHERE C > 5
GROUP BY A
HAVING SUM(C) < 20
```

- Duplication in the language leads to duplication in the code.
- It’s harder to reason about all possible combinations.

SQL: Bad for plans?

SQL is Redundant

```
SELECT A, COUNT(DISTINCT B)
FROM R
WHERE C > 5
GROUP BY A
HAVING SUM(C) < 20
```

- Duplication in the language leads to duplication in the code.
- It’s harder to reason about all possible combinations.

SQL is Order-Independent

```
SELECT R.A, T.D
FROM R, S, T
WHERE R.B = S.B
  AND S.C = T.C
```

SQL: Bad for plans?

SQL is Redundant

```
SELECT A, COUNT(DISTINCT B)
FROM R
WHERE C > 5
GROUP BY A
HAVING SUM(C) < 20
```

- Duplication in the language leads to duplication in the code.
- It’s harder to reason about all possible combinations.

SQL is Order-Independent

```
SELECT R.A, T.D
FROM R, S, T
WHERE R.B = S.B
  AND S.C = T.C
```

- Order of operations has a huge effect on performance.
- Reordering operations can create opportunities to “inline” pairs of operators.

SQL: Good for plans?

SQL is Composable

```
SELECT *
FROM R, S, (
  SELECT C, SUM(D) FROM T GROUP BY C
) T,
WHERE R.B = S.B AND S.C = T.C
```

SQL: Good for plans?

SQL is Composable

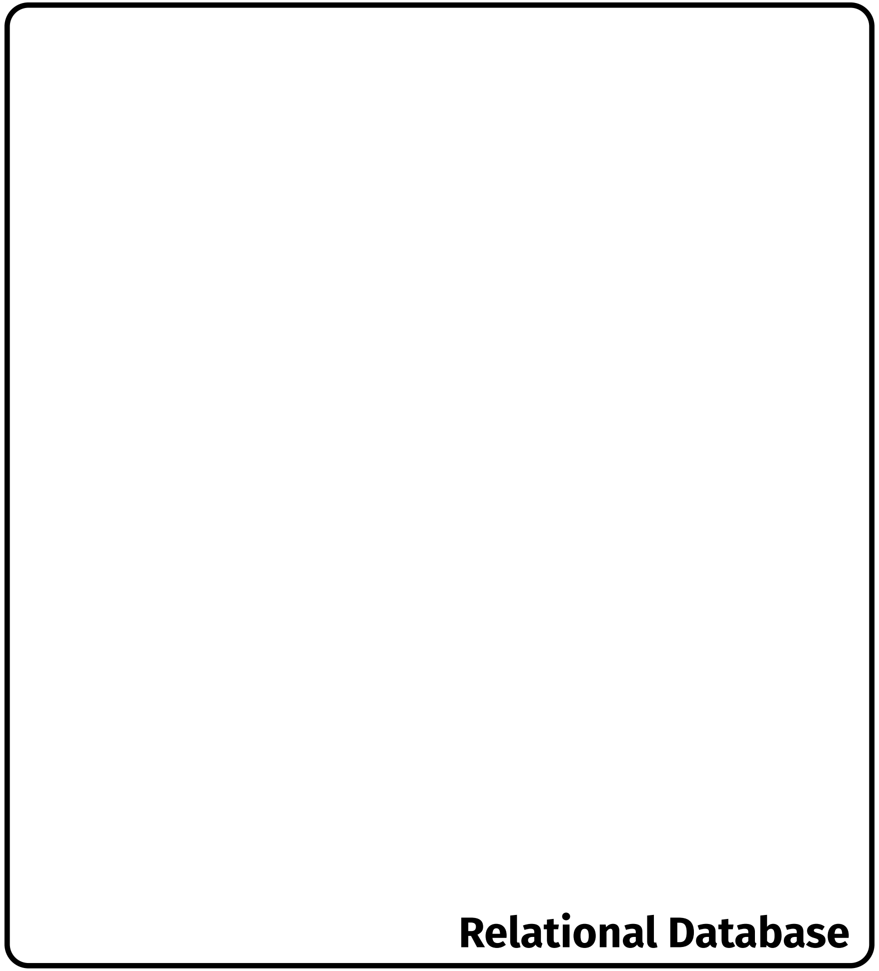
```
SELECT *
FROM R, S, (
  SELECT C, SUM(D) FROM T GROUP BY C
) T,
WHERE R.B = S.B AND S.C = T.C
```

- Queries define relations: You can use a query in place of any relation.

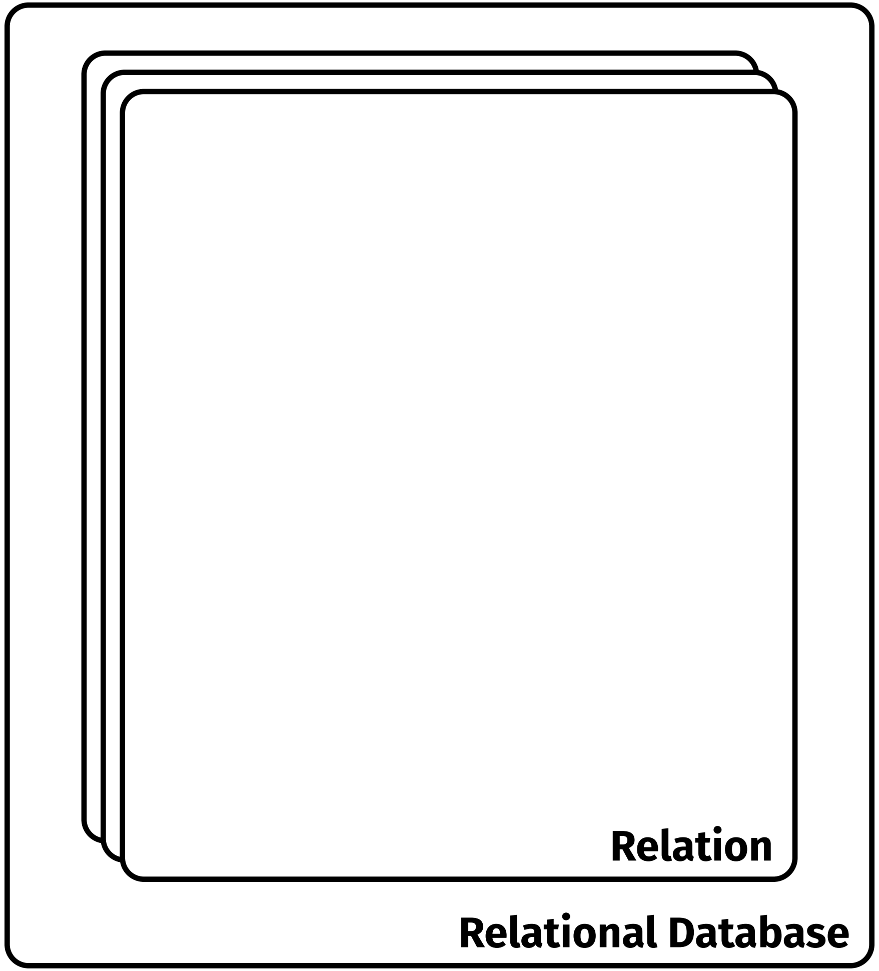
Coming up with a planning language...

1. The language should not be redundant.
2. The language should include order of operations
3. The language should be made of composable building blocks.

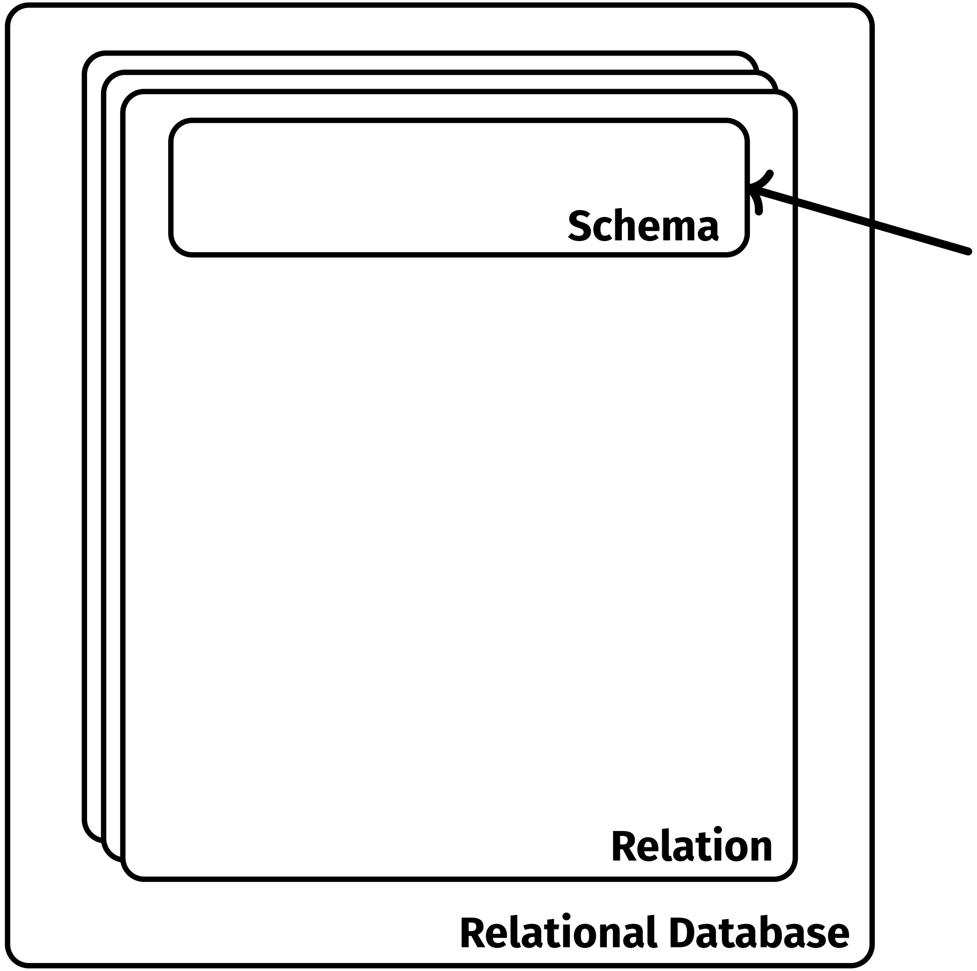
Relational Databases



Relational Databases



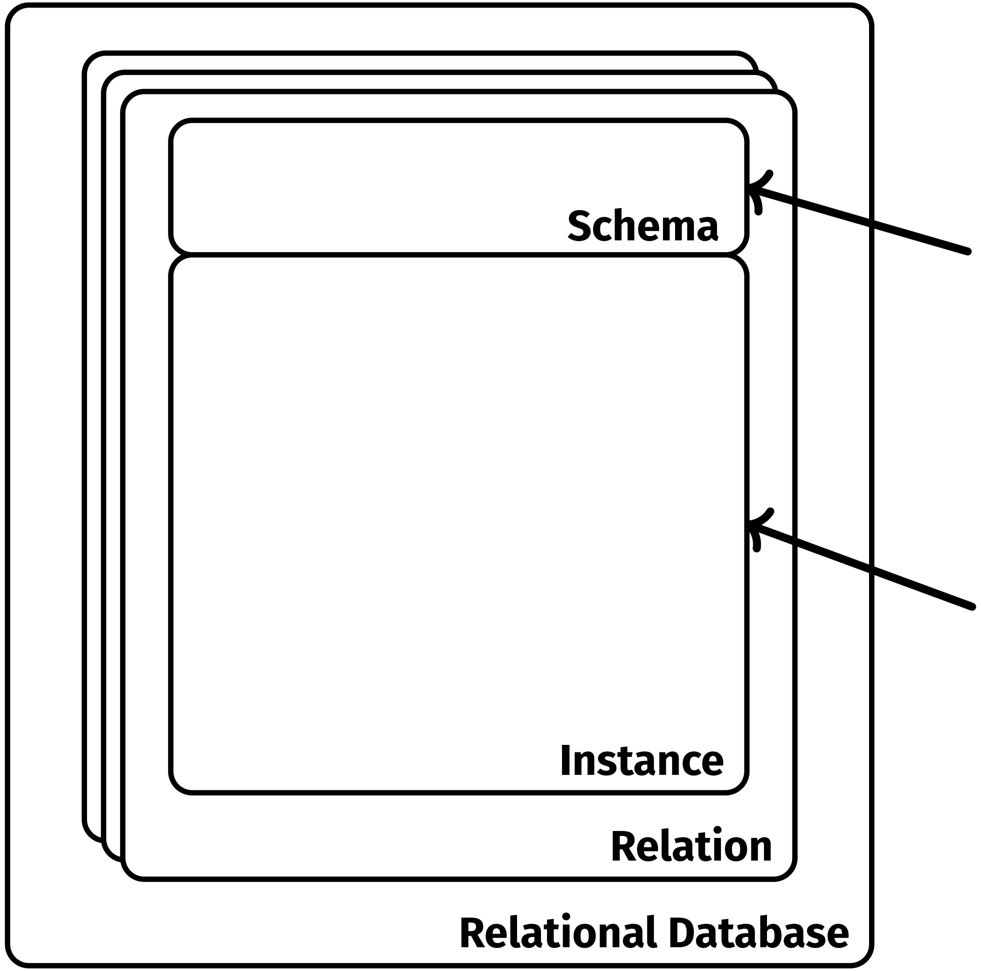
Relational Databases



Specifies the name of the relation, name and type of each column, and any other constraints

```
Officers(
  firstname string,
  lastname string,
  id int
)
```

Relational Databases



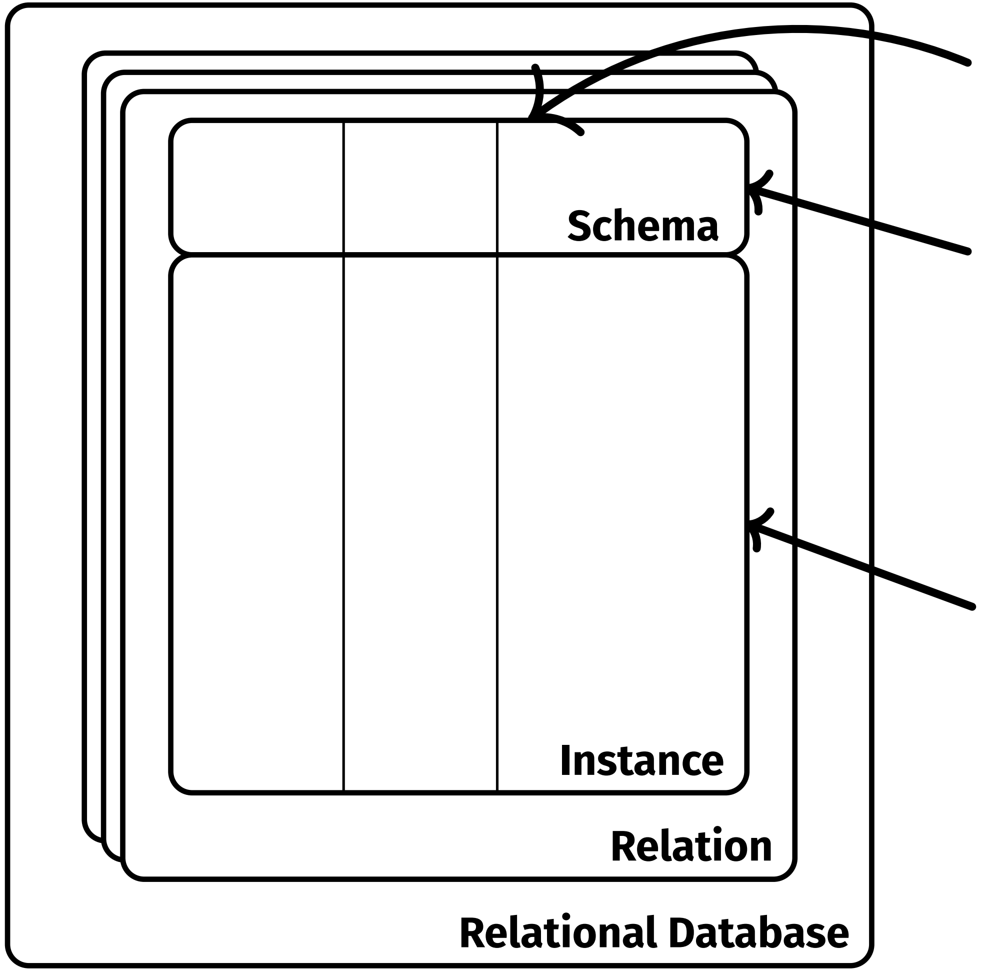
Specifies the name of the relation, name and type of each column, and any other constraints

```
Officers(
  firstname string,
  lastname string,
  id int
)
```

The Data

```
[Jean Luc, Picard, 2360]
[Benjamin, Sisko, 2365]
[Kathryn, Janeway, 2370]
```

Relational Databases



Columns (degree/arity)

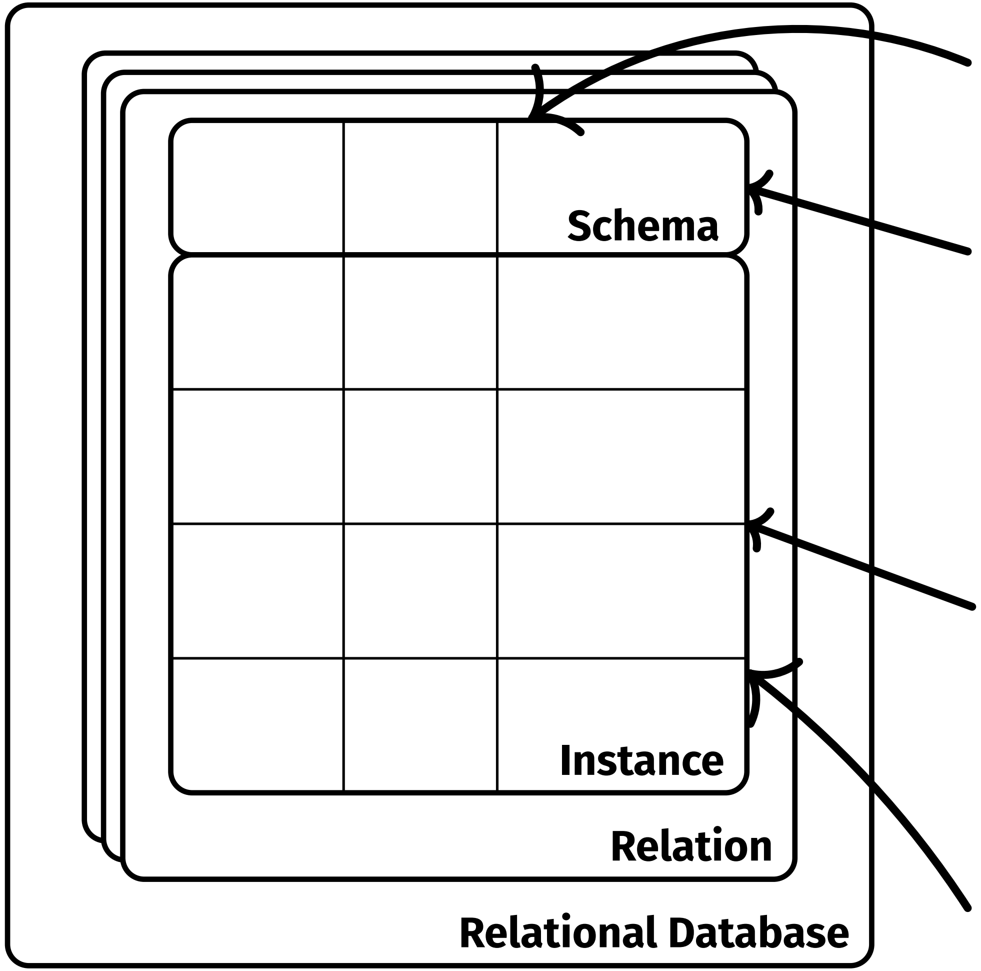
Specifies the name of the relation, name and type of each column, and any other constraints

```
Officers(
  firstname string,
  lastname string,
  id int
)
```

The Data

```
[Jean Luc, Picard, 2360]
[Benjamin, Sisko, 2365]
[Kathryn, Janeway, 2370]
```

Relational Databases



Columns (degree/arity)

Specifies the name of the relation, name and type of each column, and any other constraints

```
Officers(
  firstname string,
  lastname string,
  id int
)
```

The Data

```
[Jean Luc, Picard, 2360]
[Benjamin, Sisko, 2365]
[Kathryn, Janeway, 2370]
```

Rows (cardinality)

Composability

Everything is a relation

- Q(R)

Composability

Everything is a relation

- Q(R)
- Q(Q(R))

Composability

Everything is a relation

- Q(R)
- Q(Q(R))
- Q(Q(Q(R)))

Composability

Everything is a relation

- Q(R)
- Q(Q(R))
- Q(Q(Q(R)))
- Q Q moar.

Composability

Everything is a relation

- Q(R)
- Q(Q(R))
- Q(Q(Q(R)))
- Q Q moar.

A query language with this property is **closed**.

Composability

Everything is a relation

- Q(R)
- Q(Q(R))
- Q(Q(Q(R)))
- Q Q moar.

A query language with this property is **closed**.

Simplifying Assumptions

- All attributes have **unique** names.
- Each instance is a Bag, Set or List

Composability

Everything is a relation

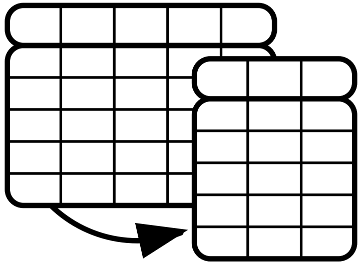
- Q(R)
- Q(Q(R))
- Q(Q(Q(R)))
- Q Q moar.

A query language with this property is **closed**.

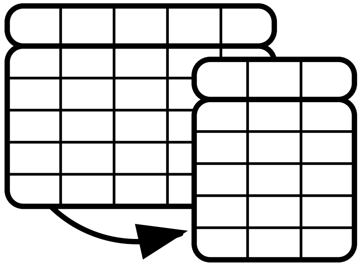
Simplifying Assumptions

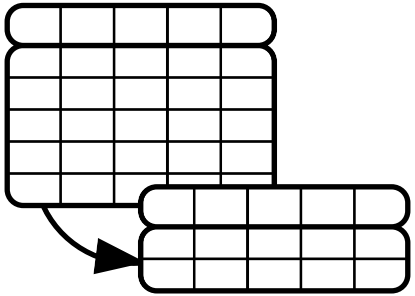
- All attributes have **unique** names.
- Each instance is a Bag, ~~Set, or List~~

Feature Desiderata

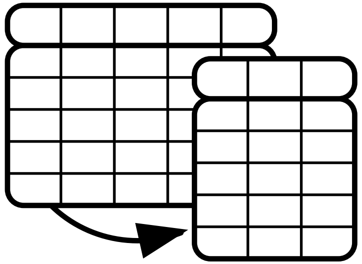


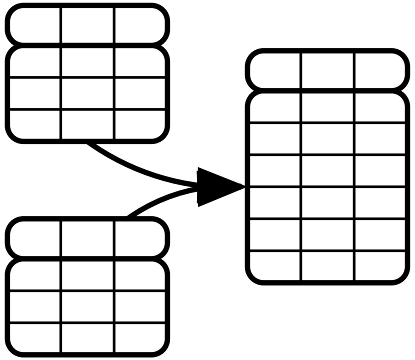
Feature Desiderata

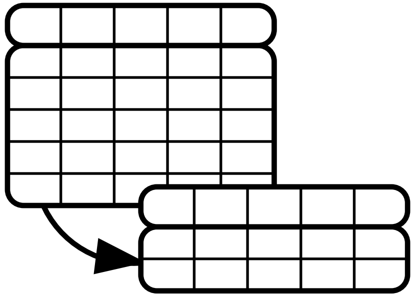




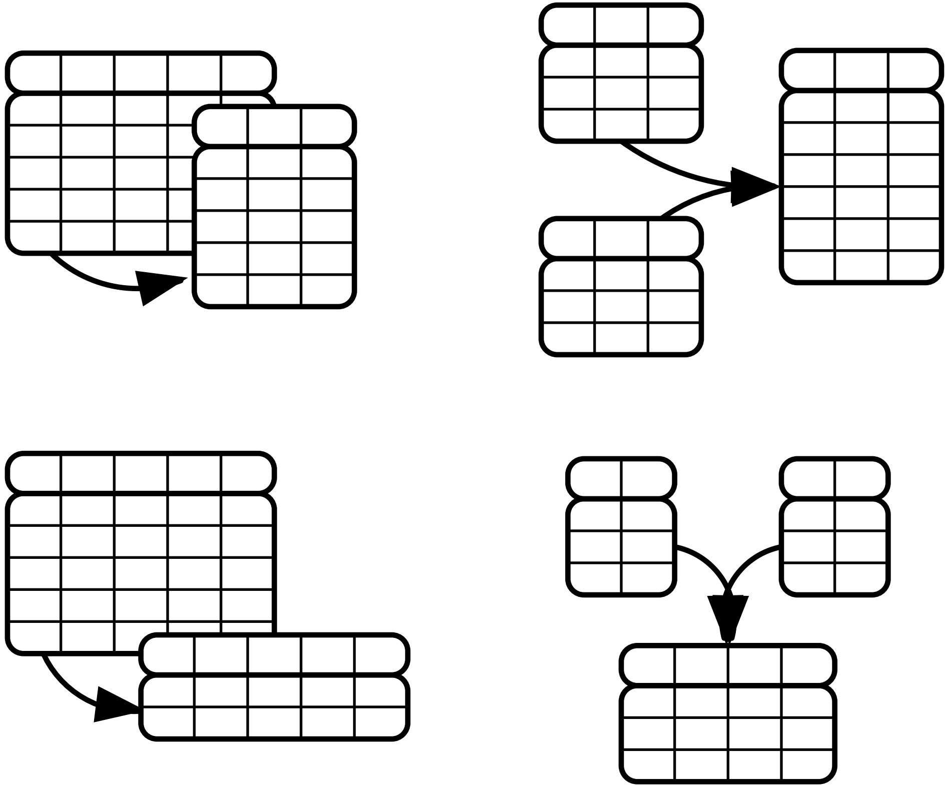
Feature Desiderata



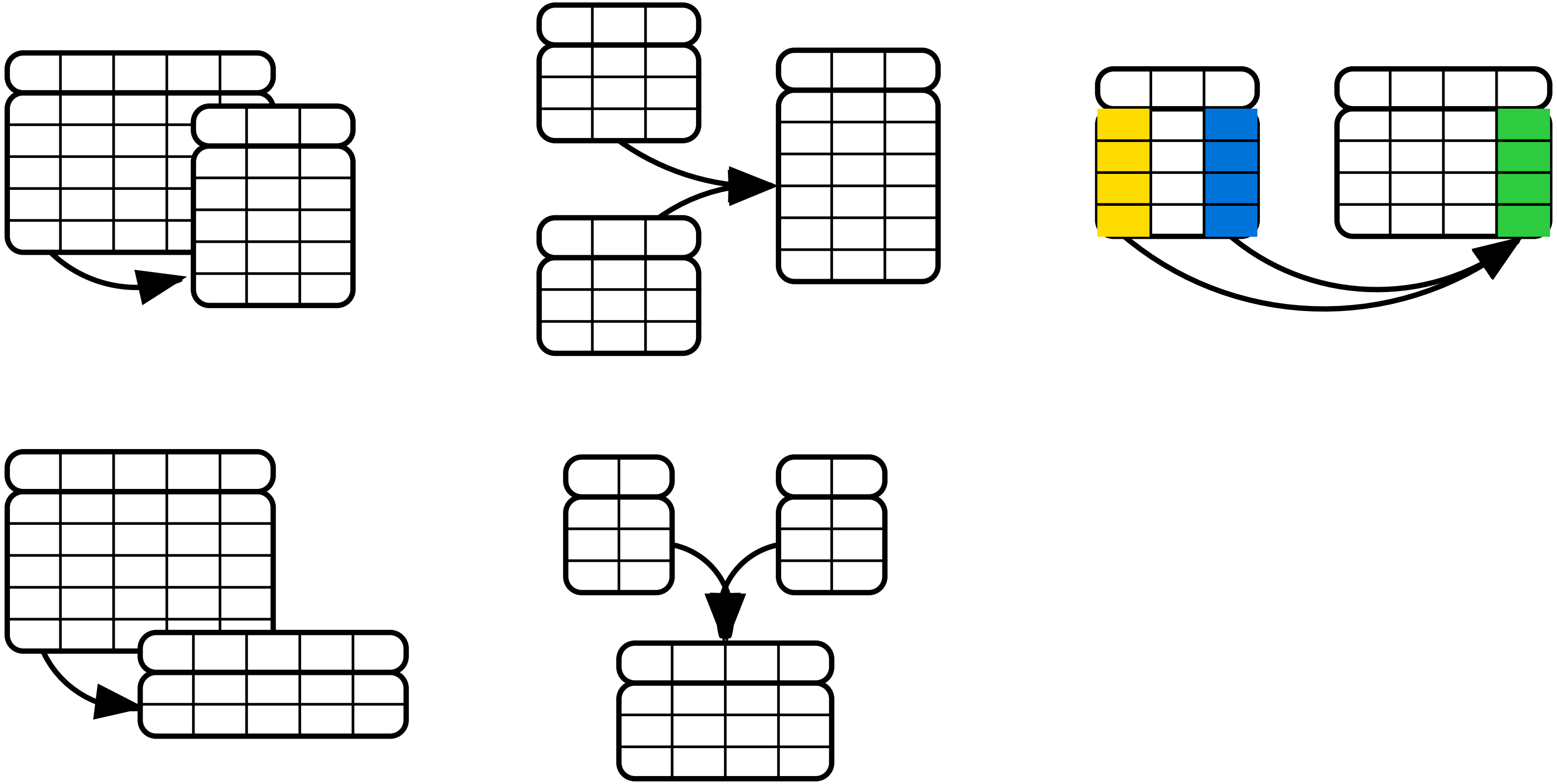




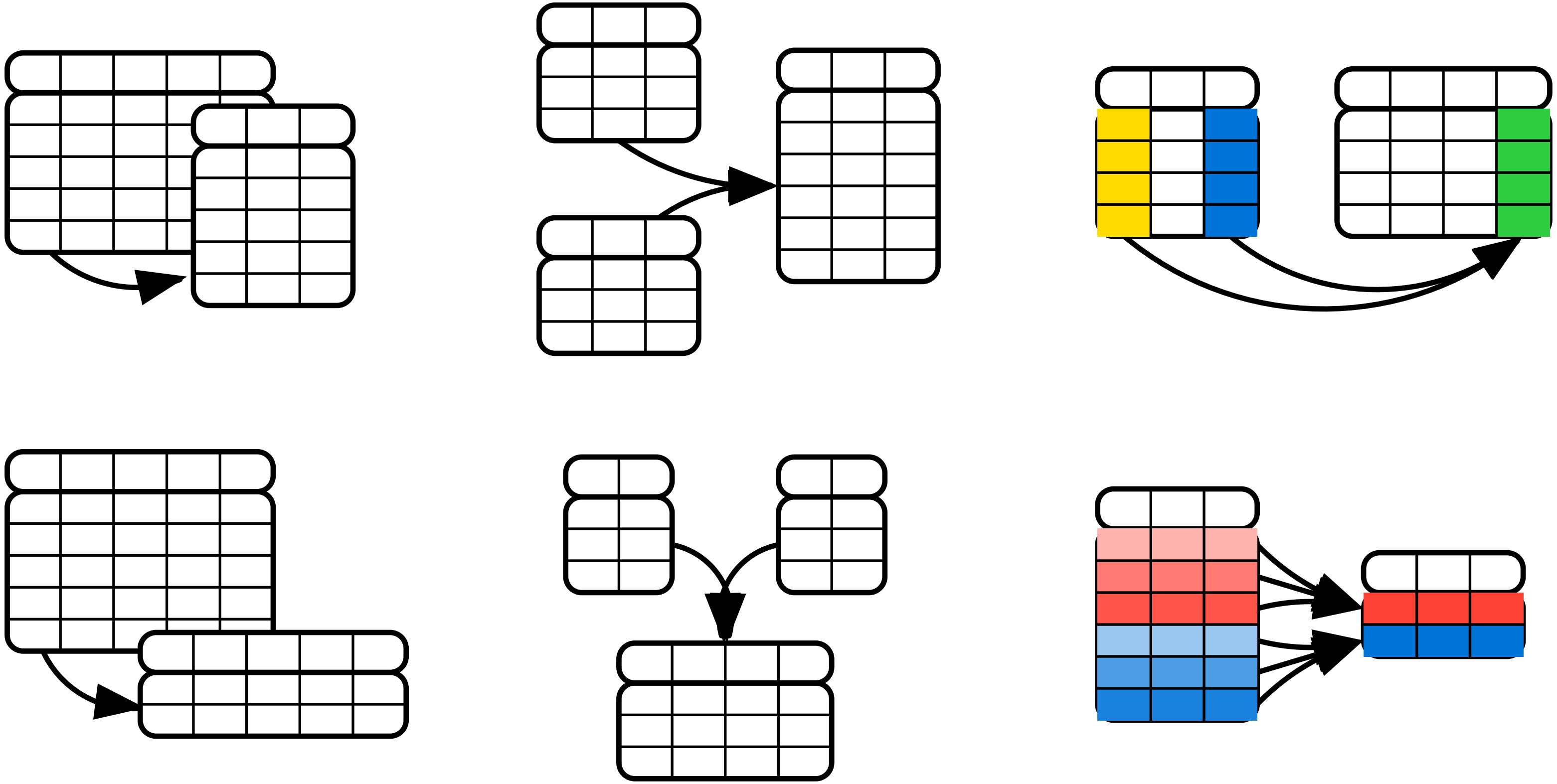
Feature Desiderata



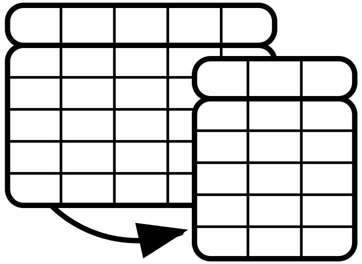
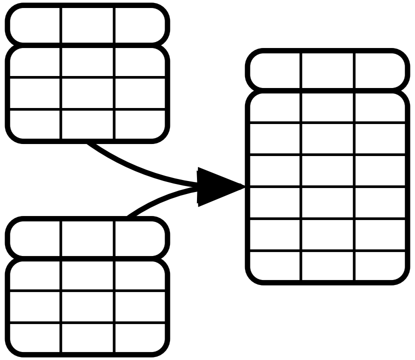
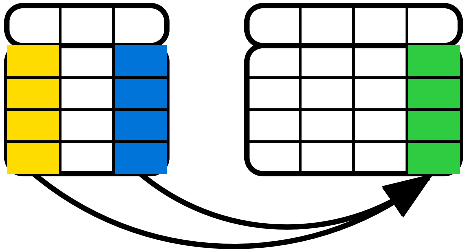
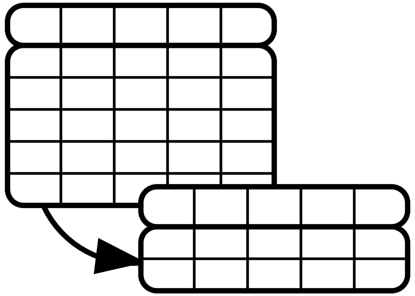
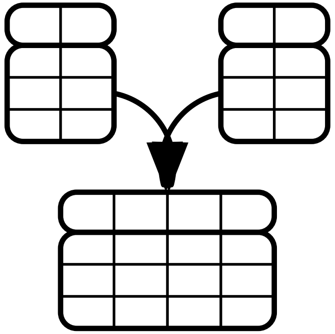
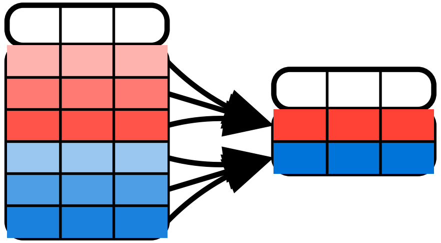
Feature Desiderata



Feature Desiderata



Feature Desiderata

	Filter	Merge	Derive
Column			
Row			

Feature Check

SQL Field	Operation
`SELECT`	
`FROM`	
`JOIN`	
`WHERE`	
`GROUP, BY`/Aggregate	
`HAVING`	

Feature Check

SQL Field	Operation
SELECT	Filter Column + Derive Column
FROM	
JOIN	
WHERE	
GROUP, BY/Aggregate	
HAVING	

Feature Check

SQL Field	Operation
`SELECT`	Filter Column + Derive Column
`FROM`	Merge Column
`JOIN`	
`WHERE`	
`GROUP, BY`/Aggregate	
`HAVING`	

Feature Check

SQL Field	Operation
`SELECT`	Filter Column + Derive Column
`FROM`	Merge Column
`JOIN`	Merge Column + Filter Row
`WHERE`	
`GROUP, BY`/Aggregate	
`HAVING`	

Feature Check

SQL Field	**Operation**
`SELECT`	Filter Column + Derive Column
`FROM`	Merge Column
`JOIN`	Merge Column + Filter Row
`WHERE`	Filter Row
`GROUP, BY`/Aggregate	
`HAVING`	

Feature Check

SQL Field	Operation
`SELECT`	Filter Column + Derive Column
`FROM`	Merge Column
`JOIN`	Merge Column + Filter Row
`WHERE`	Filter Row
`GROUP, BY`/Aggregate	Derive Row
`HAVING`	

Feature Check

SQL Field	Operation
`SELECT`	Filter Column + Derive Column
`FROM`	Merge Column
`JOIN`	Merge Column + Filter Row
`WHERE`	Filter Row
`GROUP, BY`/Aggregate	Derive Row
`HAVING`	Derive Row + Filter Row

Columns and rows are different

Columns vs Rows

Columns

- There is a “small” number (10,000s at most).
- Query planning knows everything about columns.
- Identified explicitly (by name or position)

Rows

- There is a “large” number (Millions, Billions, More)
- Query planning has only statistics (if it has anything).
- Sets/Bags have no explicit identity (Each tuple’s attributes identify it)[1]

[1] Technically false: Many DB systems have RowIDs... but these are not intended for the user

Filtering

Design Questions

- How do we indicate which columns/rows to keep/discard?

Filtering Rows

- ???

Filtering Columns

- ???

Filtering

Design Questions

- How do we indicate which columns/rows to keep/discard?

Filtering Rows

- ???

Filtering Columns

- Explicit list of columns to keep

Filtering

Design Questions

- How do we indicate which columns/rows to keep/discard?

Filtering Rows

- Condition, or “Predicate” for which rows to keep

Filtering Columns

- Explicit list of columns to keep

Filtering

Design Questions

- How do we indicate which columns/rows to keep/discard?

Filtering Rows

- Condition, or “Predicate” for which rows to keep

Filtering Columns

- Explicit list of columns to keep

Language

Code	Meaning
`Filter($a = b$, In)`	Relation `In`, keeping only rows where $a = b$
`Project([a, b, c], In)`	Relation `In`, keeping only columns a, b, c

Filtering

Design Questions

- How do we indicate which columns/rows to keep/discard?

Filtering Rows

- Condition, or “Predicate” for which rows to keep

Filtering Columns

- Explicit list of columns to keep

Language

Code	Shorthand	Meaning
`Filter($a = b$, In)`	$\sigma_{a=b}(\mathrm{In})$	Relation `In`, keeping only rows where $a = b$
`Project([a, b, c], In)`	$\pi_{a,b,c}(\mathrm{In})$	Relation `In`, keeping only columns a, b, c

Merging Rows

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Rows

- ???

Merging Rows

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Rows

- Pair columns of the same name

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Rows

- Pair columns of the same name
  - Relations must be “Union-compatible” (same schema) to be paired.
  - Some engines (e.g., Apache Spark) implicitly add columns of nulls.

Merging Rows

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Rows

- Pair columns of the same name
  - Relations must be “Union-compatible” (same schema) to be paired.
  - Some engines (e.g., Apache Spark) implicitly add columns of nulls.

Language

Code	Meaning
`Union(In1, In2)`	All rows from both relations In_1 and In_2

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Rows

- Pair columns of the same name
  - Relations must be “Union-compatible” (same schema) to be paired.
  - Some engines (e.g., Apache Spark) implicitly add columns of nulls.

Language

Code	Shorthand	Meaning
`Union(In1, In2)`	$\mathrm{In}_1 \cup \mathrm{In}_2$	All rows from both relations In_1 and In_2

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Columns

- ???

Merging Columns

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Columns

- Pair all rows

Merging Columns

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Columns

- Pair all rows
  - Use filter to keep only the rows you want

Merging Columns

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Columns

- Pair all rows
  - Use filter to keep only the rows you want

Language

Code	Meaning
`Product(In1, In2)`	Every possible pair of rows from $\mathrm{In}_1, \mathrm{In}_2$

Merging Columns

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Columns

- Pair all rows
  - Use filter to keep only the rows you want

Language

Code	Shorthand	Meaning
`Product(In1, In2)`	$\mathrm{In}_1 \times \mathrm{In}_2$	Every possible pair of rows from In_1, In_2

Merging Columns

Design Questions

- Which rows/columns pair with which other rows/columns?

Merging Columns

- Pair all rows
  - Use filter to keep only the rows you want

Language

Code	Shorthand	Meaning
`Product(In1, In2)`	$\mathrm{In}_1 \times \mathrm{In}_2$	Every possible pair of rows from In_1, In_2
`Join($a = b$, In1, In2)`	$\mathrm{In}_1 \underset{a=b}{\bowtie} \mathrm{In}_2$	Shorthand for $\sigma_{a=b}(\mathrm{In}_1 \times \mathrm{In}_2)$

A note on join terminology...

Cartesian Product

- $R \times S$: Every pair of one tuple from R and S

Join

- $R \underset{a>b}{\bowtie} S$: Only include pairs where the predicate $a > b$ is true

Equi-Join

- $R \underset{a=b}{\bowtie} S$: A join that only uses equality predicates (e.g., $a = b \land c = d$)
- $R \underset{a}{\bowtie} S$: If the equi-join columns have the same name, we just write the name(s)

Natural Join

- $R \bowtie S$: If the join predicate is omitted, assume an equi-join between all columns with the same name.

Derive Columns

Design Questions

- How do we specify which columns/rows to combine?
- How do we specify how to define the new row/column?

Deriving Columns

- ???

Derive Columns

Design Questions

- How do we specify which columns/rows to combine?
- How do we specify how to define the new row/column?

Deriving Columns

- `A + B AS C` or `C = A + B`

Derive Columns

Design Questions

- How do we specify which columns/rows to combine?
- How do we specify how to define the new row/column?

Deriving Columns

- `A + B AS C` or `C = A + B`

Language

Code	Meaning
`Project([A, B, C = $A+B$], In)`	As Project, but derive C

Derive Columns

Design Questions

- How do we specify which columns/rows to combine?
- How do we specify how to define the new row/column?

Deriving Columns

- `A + B AS C` or `C = A + B`

Language

Code	Shorthand	Meaning
`Project([A, B, C = $A+B$], In)`	$\pi_{A,B,C=A+B}(\mathrm{In})$	As Project, but derive C

Derive Rows

Design Questions

- How do we specify which columns/rows to combine?
- How do we specify how to define the new row/column?

Deriving Rows

- ???
- ???

Derive Rows

Design Questions

- How do we specify which columns/rows to combine?
- How do we specify how to define the new row/column?

Deriving Rows

- “Group By” attributes[2]
- ???

[2] We’ll eventually talk about other grouping strategies (e.g., Window Functions)

Derive Rows

Design Questions

- How do we specify which columns/rows to combine?
- How do we specify how to define the new row/column?

Deriving Rows

- “Group By” attributes[3]
- “Aggregate functions”

Code

```
GroupBy([A, B], [SUM(C) AS C, AVG(D) AS D], In)
```

Group tuples by A and B; For each group compute $\mathrm{SUM}(C)$, $\mathrm{AVG}(D)$

[3] We’ll eventually talk about other grouping strategies (e.g., Window Functions)

Derive Rows

Design Questions

- How do we specify which columns/rows to combine?
- How do we specify how to define the new row/column?

Deriving Rows

- “Group By” attributes[4]
- “Aggregate functions”

Code	Shorthand
`GroupBy([A, B], [SUM(C) AS C, AVG(D) AS D], In)`	$\Sigma_{A,B,C=\mathrm{SUM}(C),D=\mathrm{AVG}(D)}(\mathrm{In})$

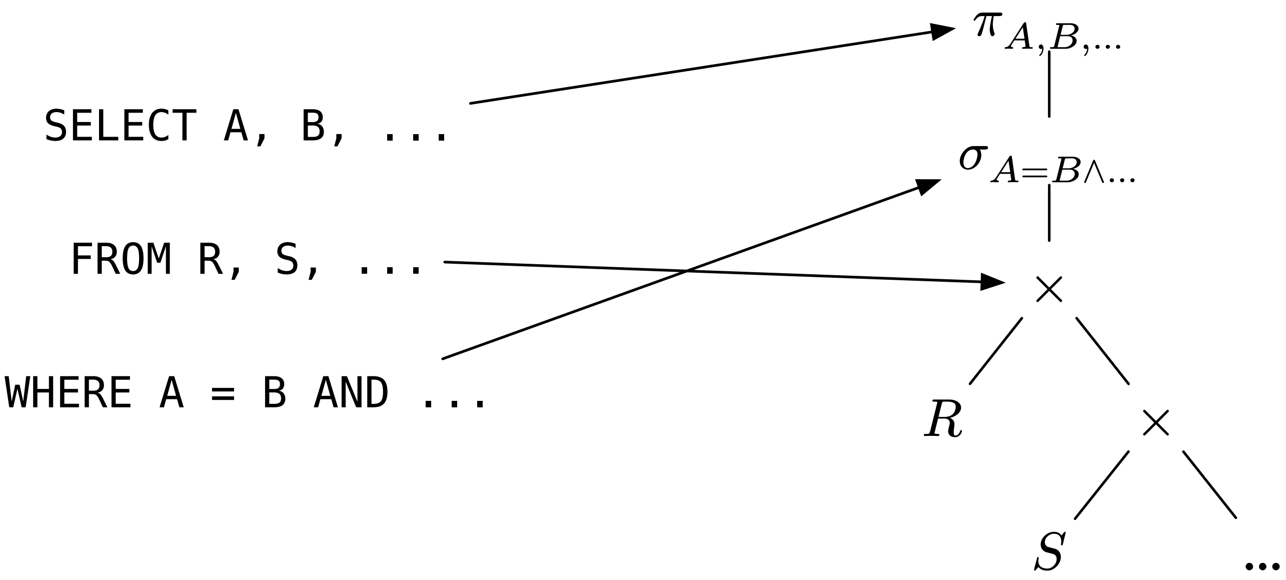
Group tuples by A and B; For each group compute $\mathrm{SUM}(C)$, $\mathrm{AVG}(D)$

[4] We’ll eventually talk about other grouping strategies (e.g., Window Functions)

Summary

	Filter	**Merge**	**Derive**
Column	π	$\times, \bowtie$	π
Row	σ	$\cup$	Σ

SQL to RA



```
SELECT A, B, ...
 FROM R, S, ...
WHERE A = B AND ...
```

Reminders

- If you have not yet formed a group **contact me!**.
- Finish the AI quiz ASAP.
- Checkpoint 1 posted and available.