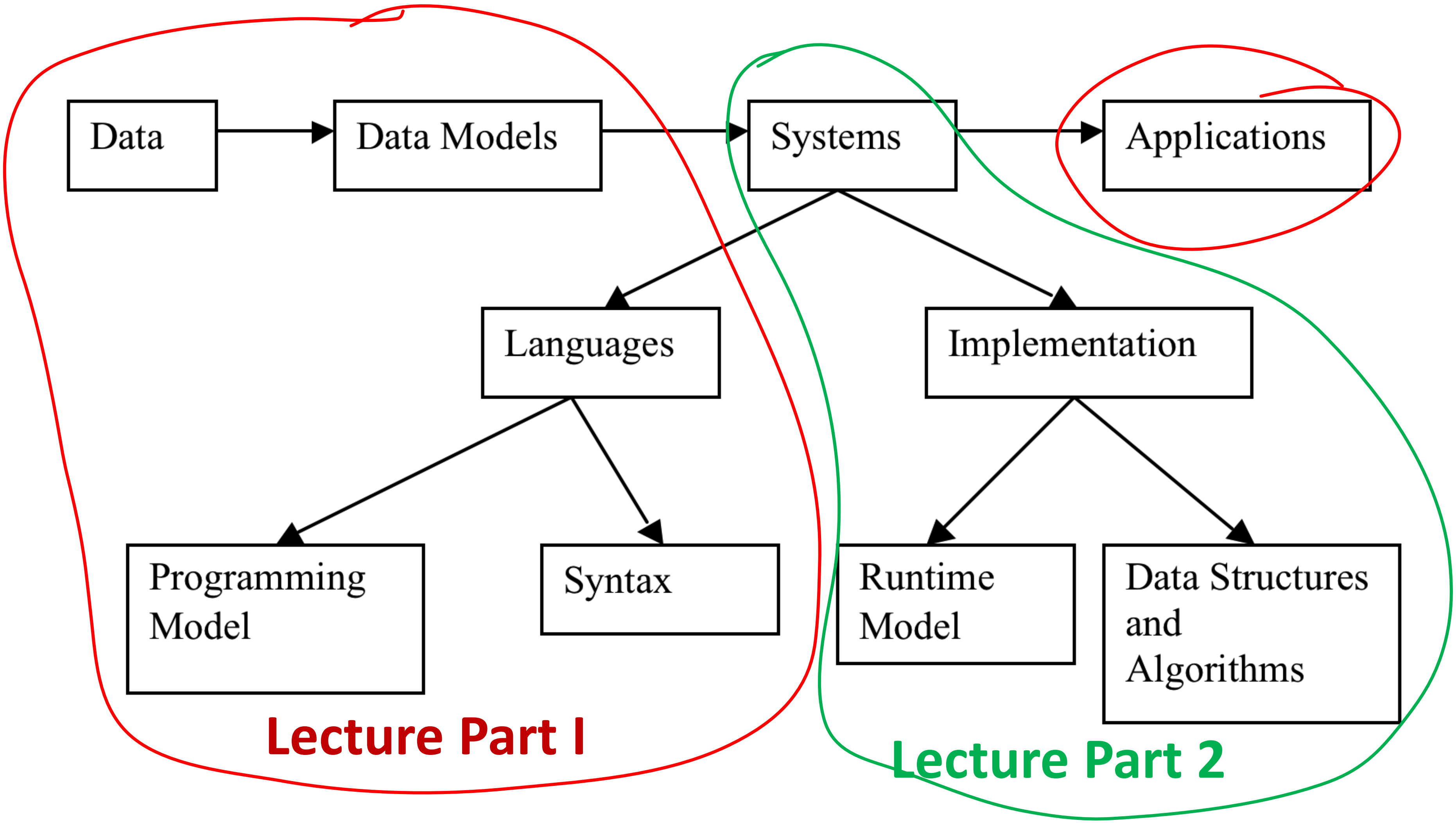
Module 4

Implementation of XQuery

Part 0: Background on relational query processing

The Data Management Universe



What does a Database System do?

- Input: SQL statement
- Output: {tuples}

1. *Translate SQL into get/put requests to backend storage*
2. *Extract, process, transform tuples from blocks*

- Tons of optimizations
  - Efficient algorithms for SQL operators (hashing, sorting)
  - Layout of data on backend storage (clustering, free space)
  - Ordering of operators (small intermediate results)
  - Semantic rewritings of queries
  - Buffer management and caching
  - Parallel execution and concurrency
  - Outsmart the OS
  - Partitioning and Replication in distributed system
  - Indexing and Materialization
  - Load and admission control

- + Security + Durability + Concurrency Control + Tools

XQuery: a mix of paradigms

- Query languages (~SQL)
- Functional programming languages (~Haskell)
- Object-oriented query languages (~OQL)
- Procedural languages (~Java)
- Some new features : context sensitive semantics

- Processing XQuery involves
  - stealing from all other languages
  - plus specific innovations

XQuery processing: old and new

Functional programming

- ☑ Environment for expressions
- ☑ Expressions nested with full generality
- ☑ Lazy evaluation
- ☒ Data Model, schemas, type system, and query language
- ☒ Contextual semantics for expressions
- ☒ Side effects
- ☒ Non-determinism in logic operations, others
- ☒ *Streaming execution*
- ☒ *Logical/physical data mismatch, appropriate optimizations*

Relational query (SQL)

- ☑ High level construct (FLWOR/Select-From-Where)
- ☑ Streaming execution
- ☑ Logical/physical data mismatch and the appropriate optimizations
- ☒ *Data Model, schemas, type system, and query language*
- ☒ *Expressive power*
- ☒ Error handling
- ☒ 2 valued logic

XQuery processing: old and new

Object-oriented query languages (OQL)

- ☑ Expressions nested with full generality
- ☑ Nodes with node/object identity
- ☒ Topological order for nodes
- ☒ *Data Model, schemas, type system, and query language*
- ☒ *Side effects*
- ☒ *Streaming execution*

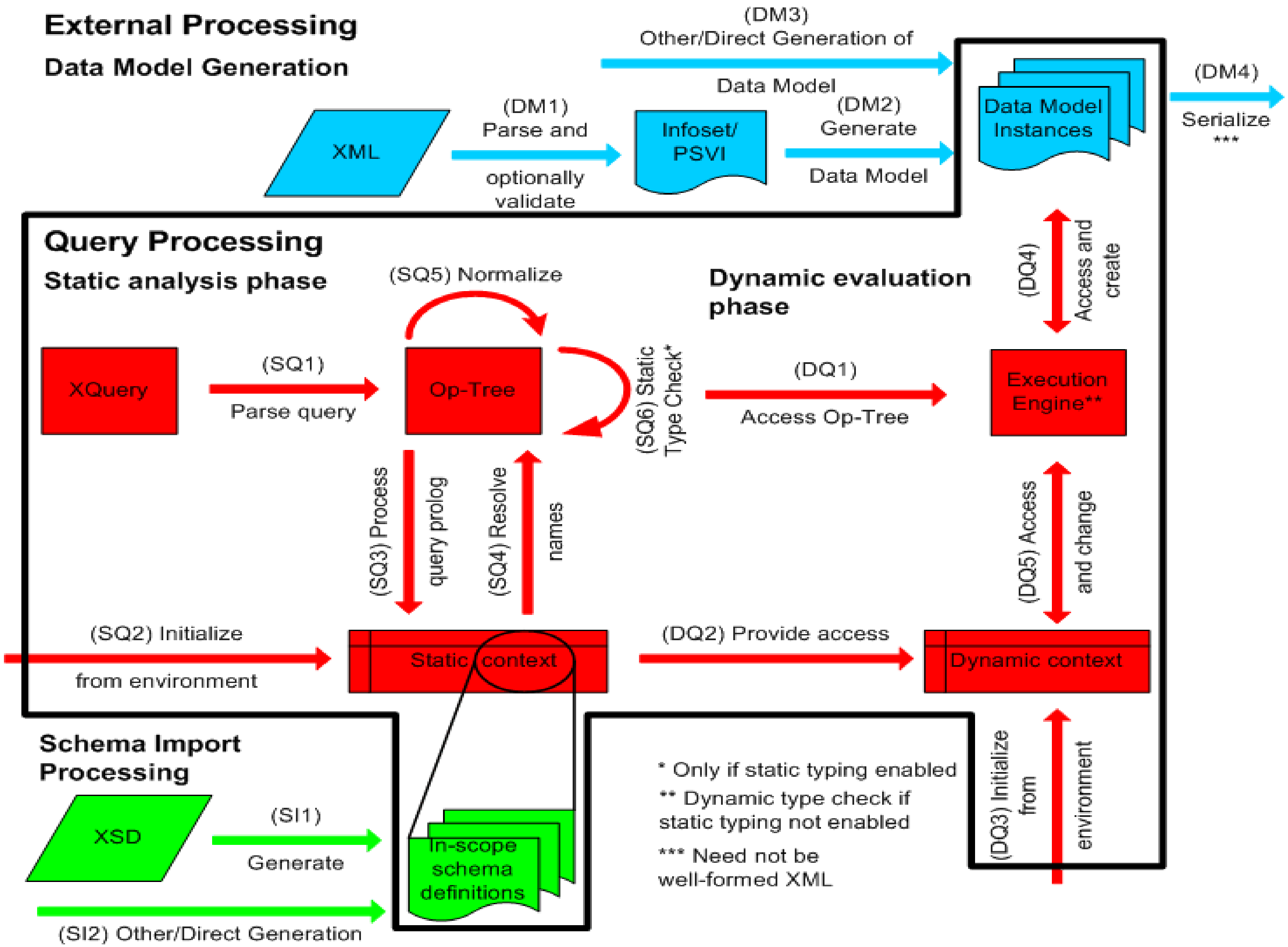
Imperative languages (e.g. Java)

- ☑ Side effects
- ☑ Error handling
- ☒ *Data Model, schemas, type system, and query language*
- ☒ Non-determinism for logic operators
- ☒ *Lazy evaluation and streaming*
- ☒ *Logical/physical data mismatch and the appropriate optimizations*
- ☒ *Possibility of handling large volumes of data*

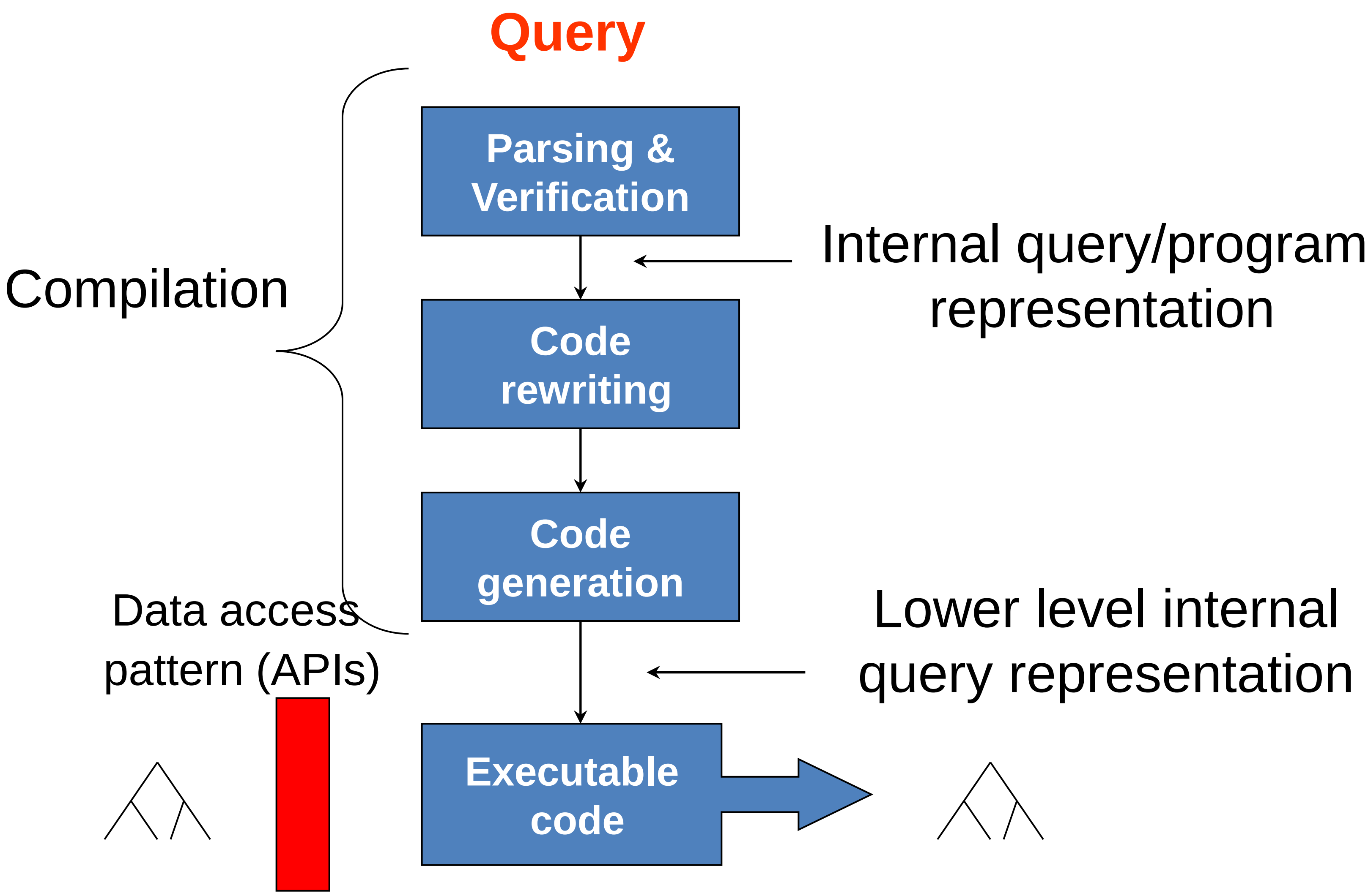
Aspects of XQuery Implementation

- Compile Time + Optimizations
  - Operator Models
  - Query Rewrite
  - Runtime + Query Execution
- XML Data Representation
  - XML Storage
  - XML Indexes
  - Compression + Binary XML

XQuery Processing Model



Architecture of (X)Query Processor



Backgrounds from the database world

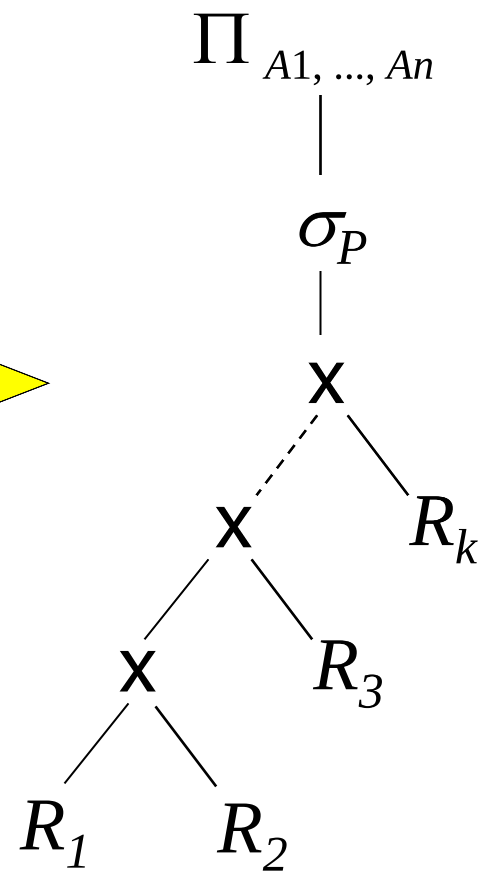
- Database management systems provides
  - a success story for building large-data, declarative infrastructures
  - Blueprints on architecture and algorithms
- No class at Uni Freiburg (explicitly) teaches these contents (as opposed to e.g., compiler construction etc)
- Quick tour through relational concepts:
  - Algebra
  - Query Processing
  - Optimization

SQL -> Relational Algebra

SQL

Relational Algebra

$$\Pi_{A1, \ldots, An}(\sigma_P (R_1 \times \ldots \times R_k))$$



select $A_1, \ldots, A_n$

from $R_1, \ldots, R_k$

where P;

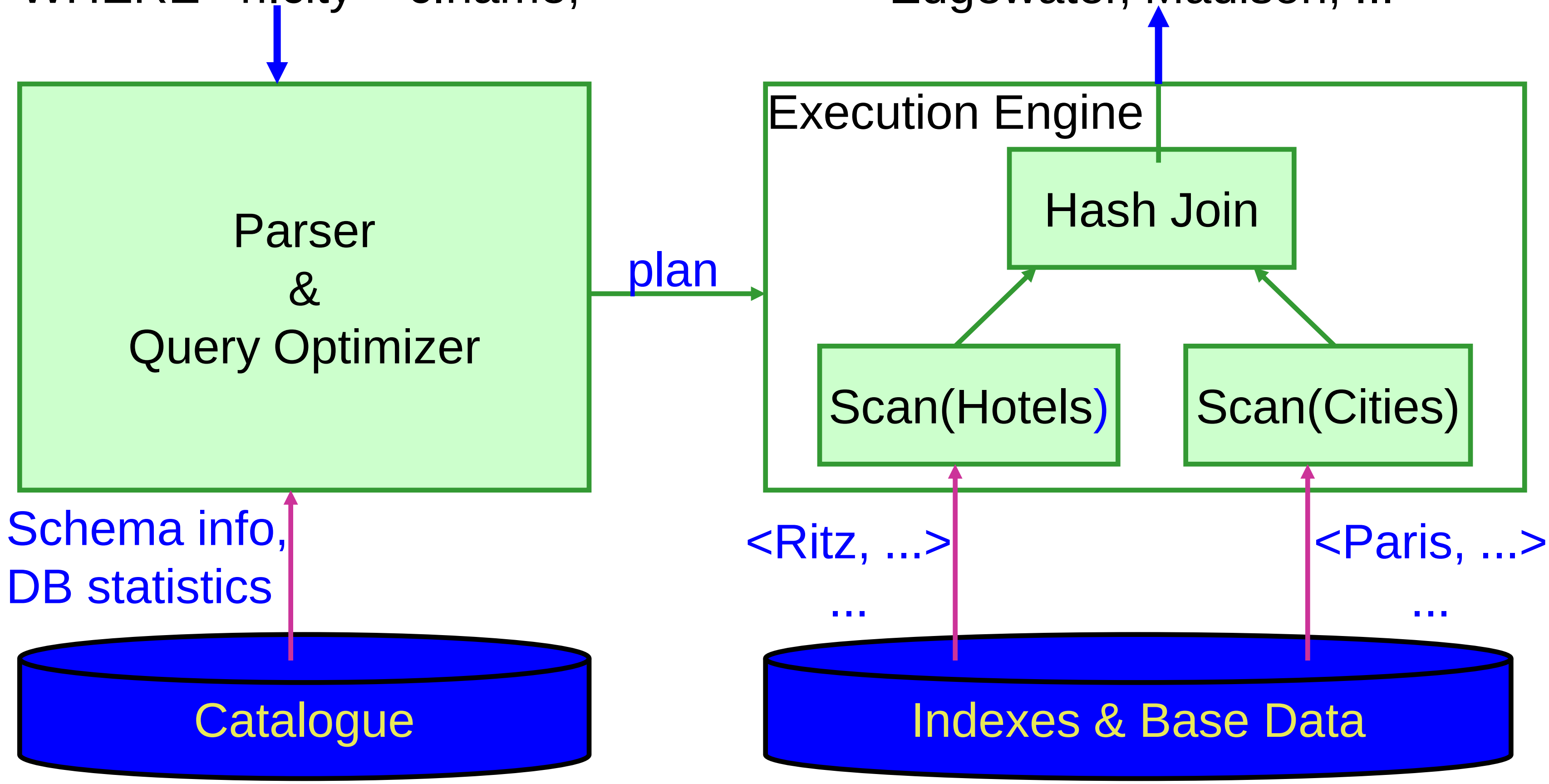
Algorithms for Rel. Algebra

- Table Access
  - scan (load each page at a time)
  - index scan (if index available)
- Sorting
  - Two-phase external sorting
- Joins
  - (Block) nested-loops
  - Index nested-loops
  - Sort-Merge
  - Hashing (many variants)
- Group-by (~ self-join)
  - Sorting, Hashing

SQL Query Processing 101

```
SELECT  *
FROM    Hotels h, Cities c
WHERE   h.city = c.name;
```

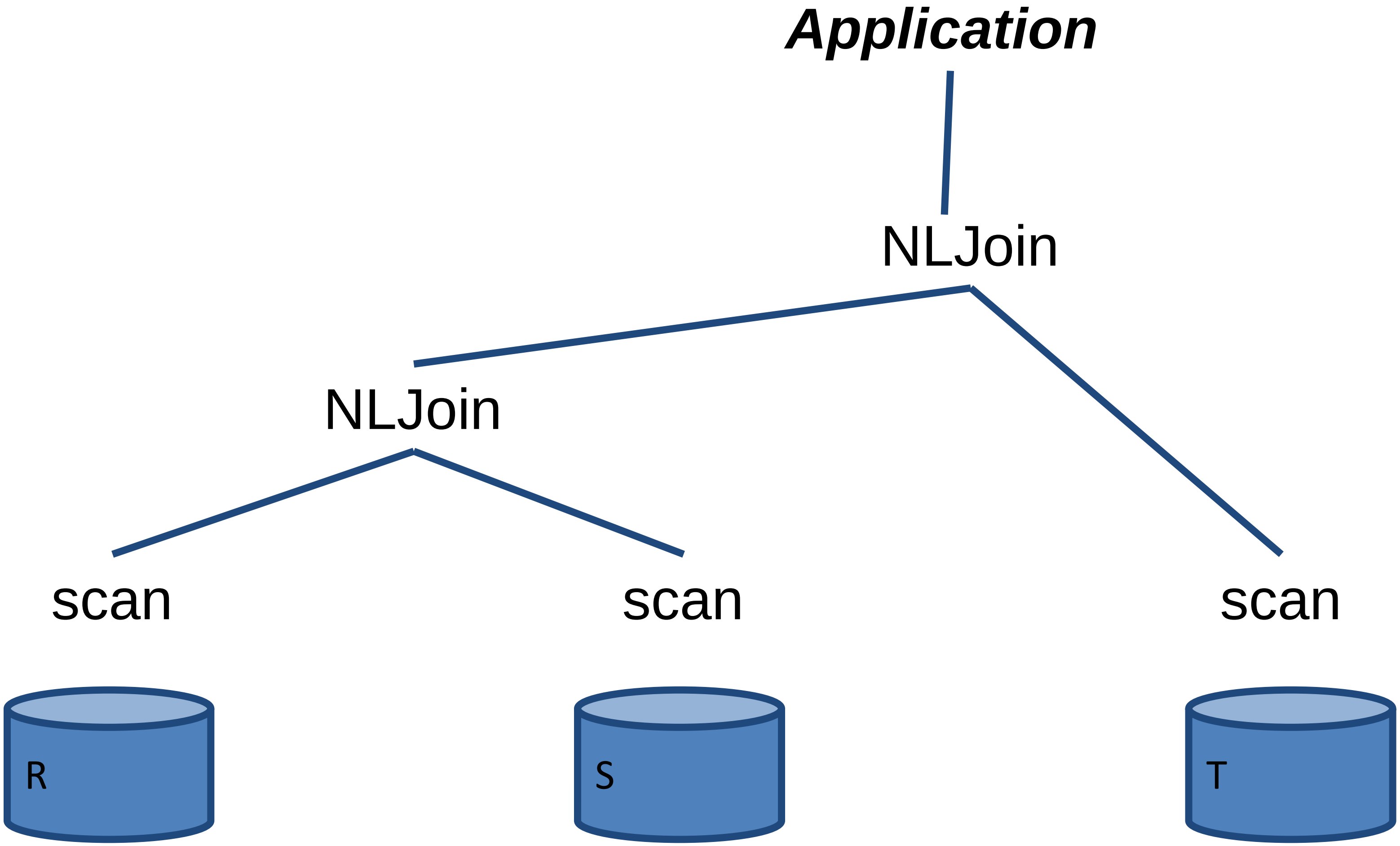
<Ritz, Paris, ...>
<Baur au Lac, Zurich, ...>
<Edgewater, Madison, ...>



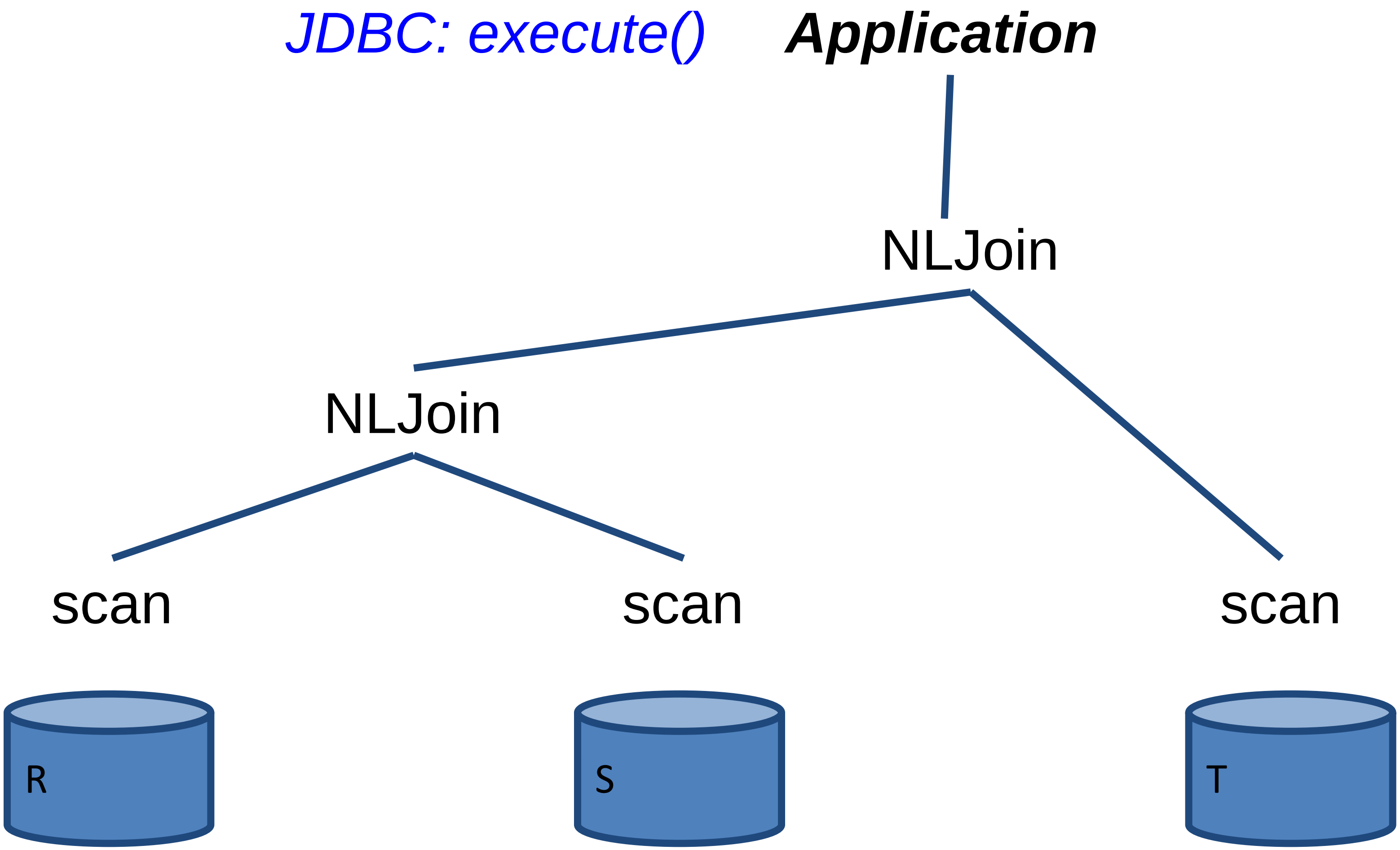
Iterator Model

- Plan contains many operators
  - Implement each operator indepently
  - Define generic interface for each operator
  - Each operator implemented by an *iterator*
- Three methods implemented by each iterator
  - open(): initialize the internal state (e.g., buffer)
  - *char* next()*: produce the next result tuple
  - *close()*: clean-up (e.g., release buffer)
- N.B. Modern DBMS use a Vector Model
  - next() returns a set of tuples
  - Why is that better?

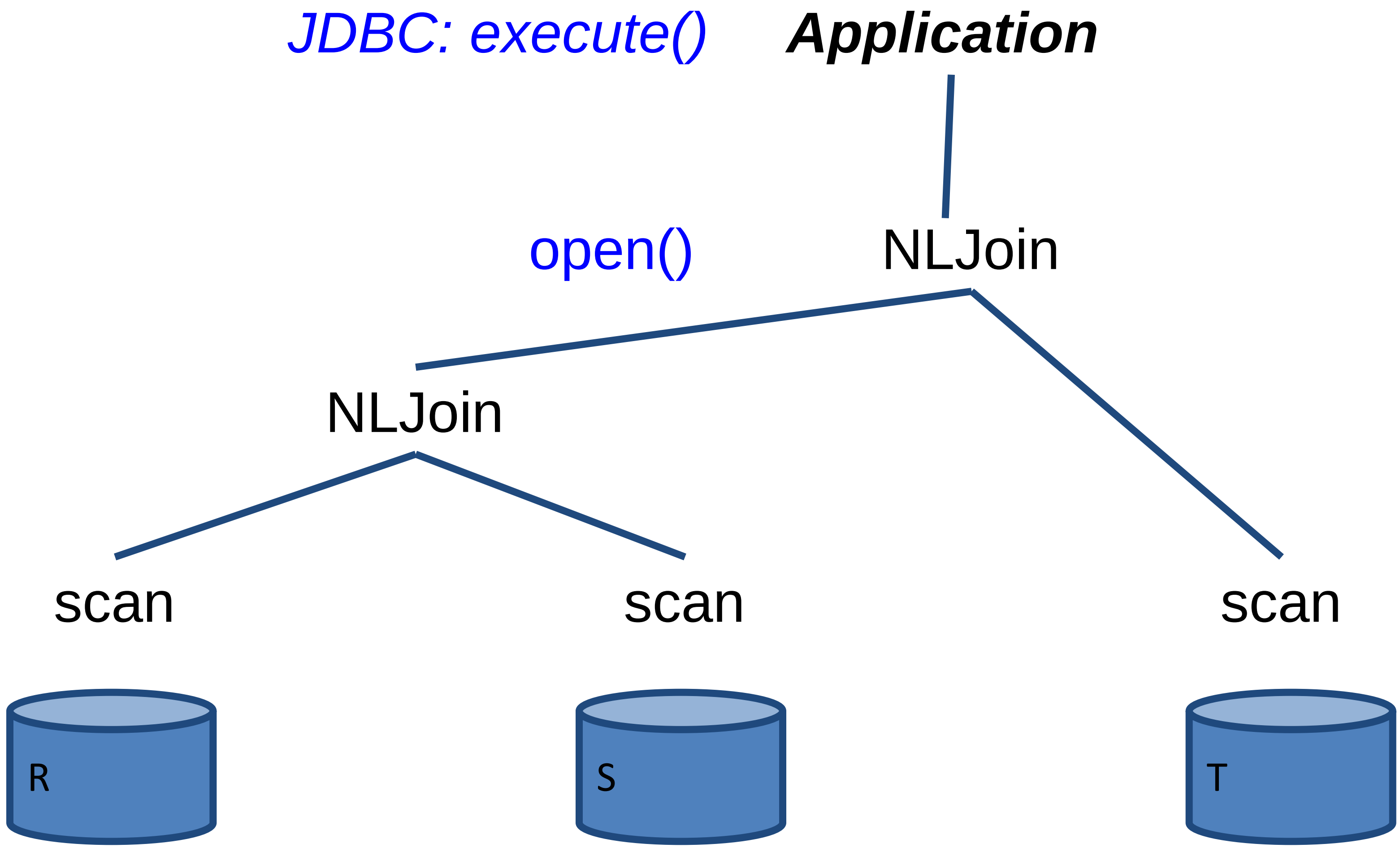
Iterator Model at Work



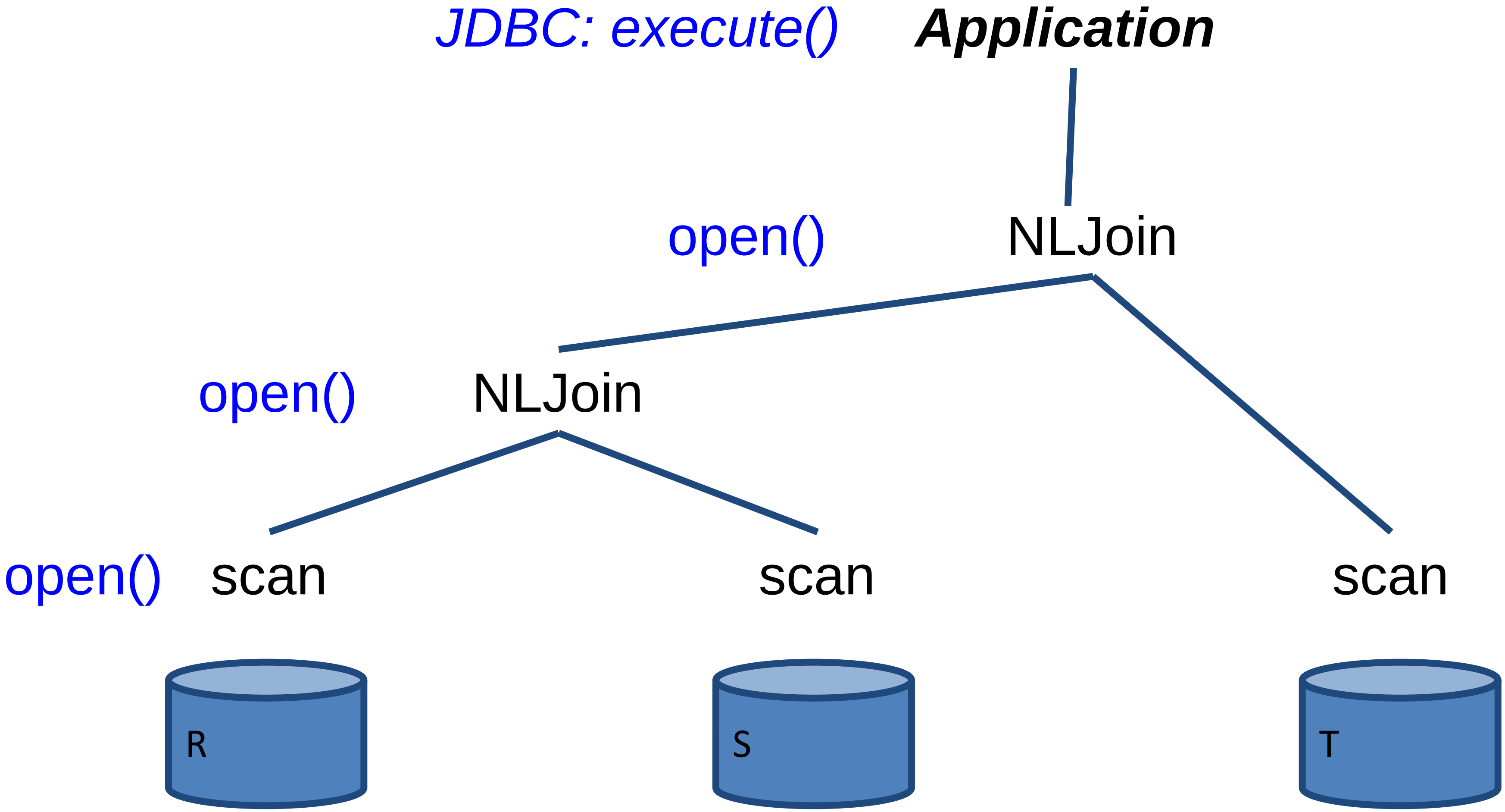
Iterator Model at Work



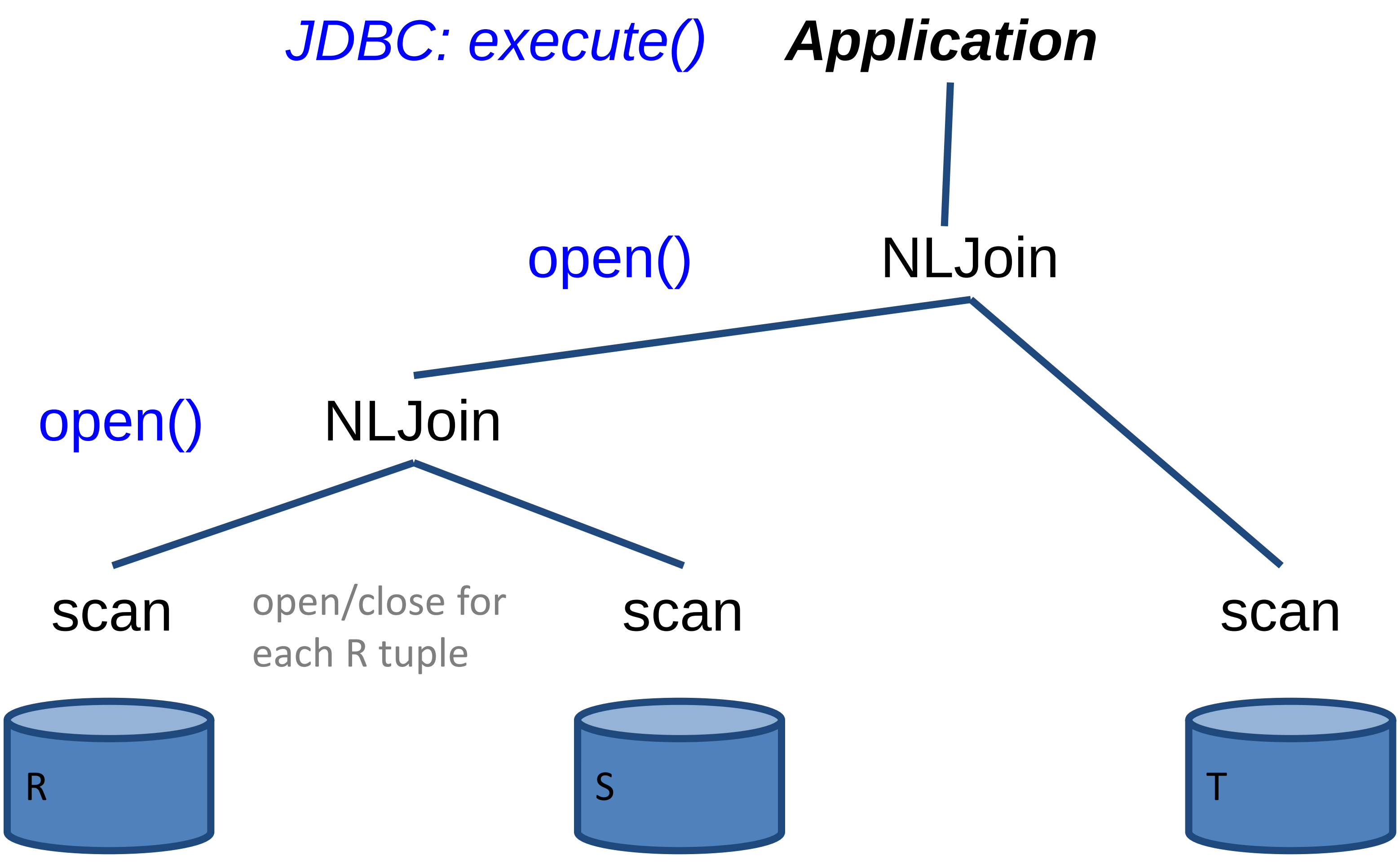
Iterator Model at Work



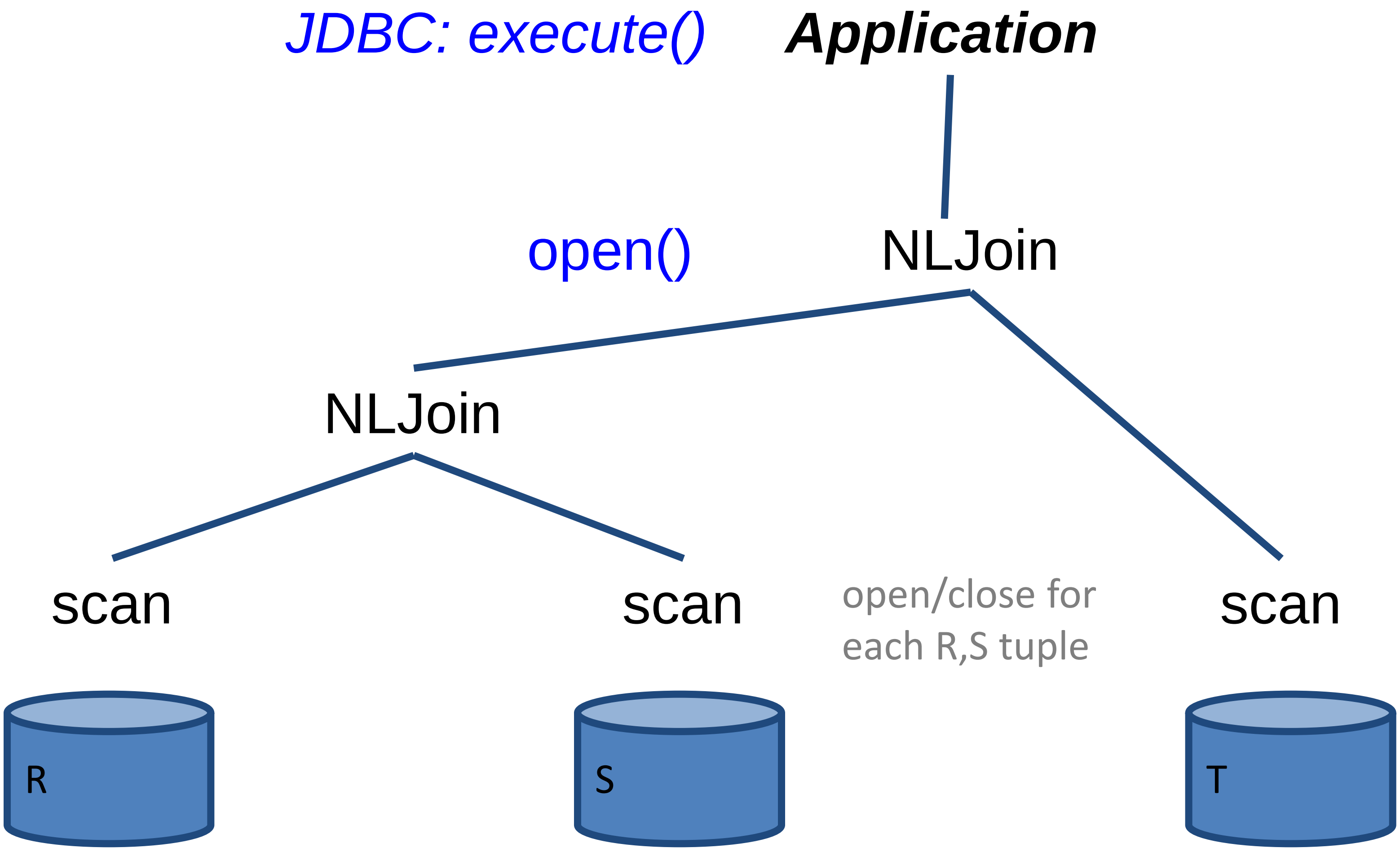
Iterator Model at Work



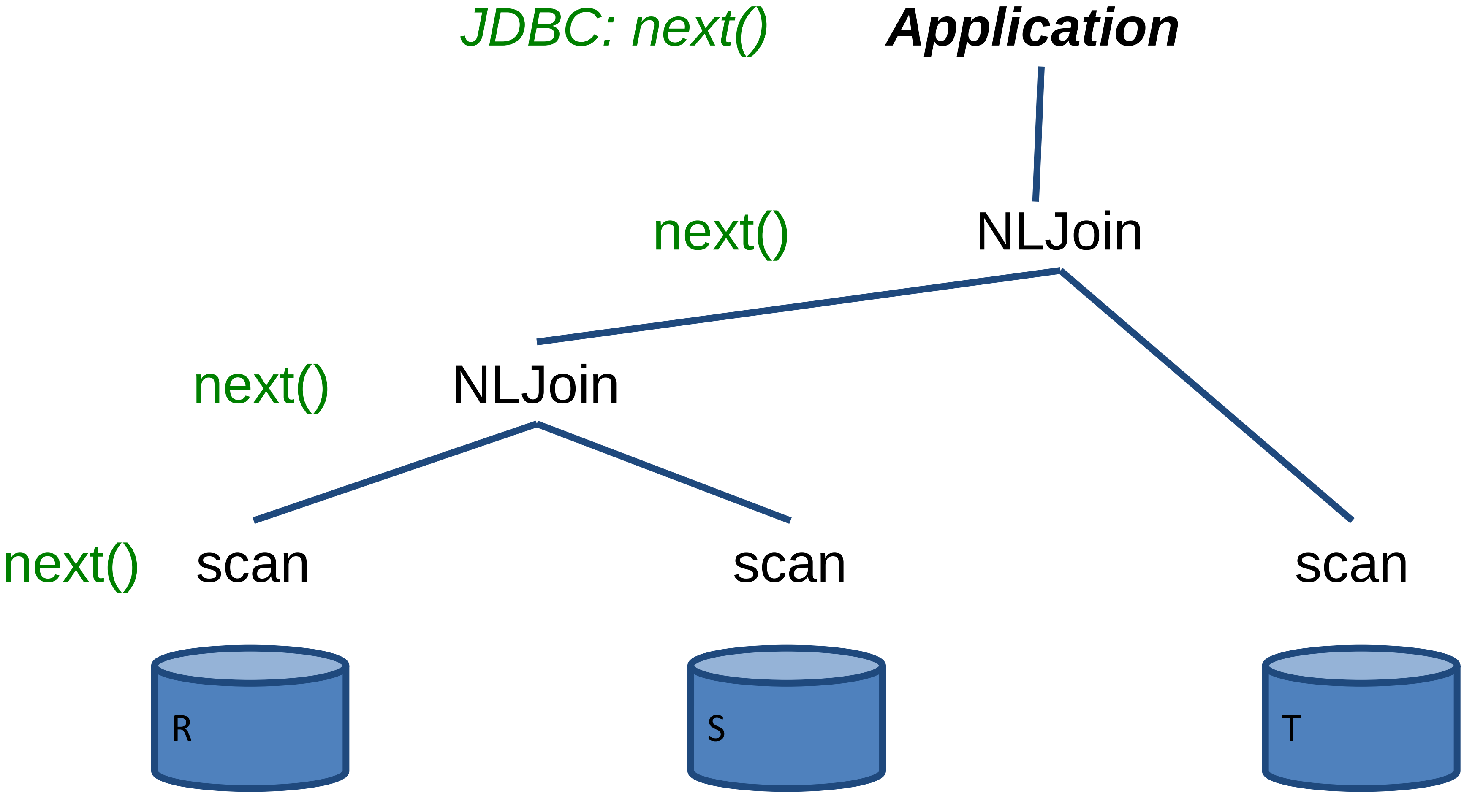
Iterator Model at Work



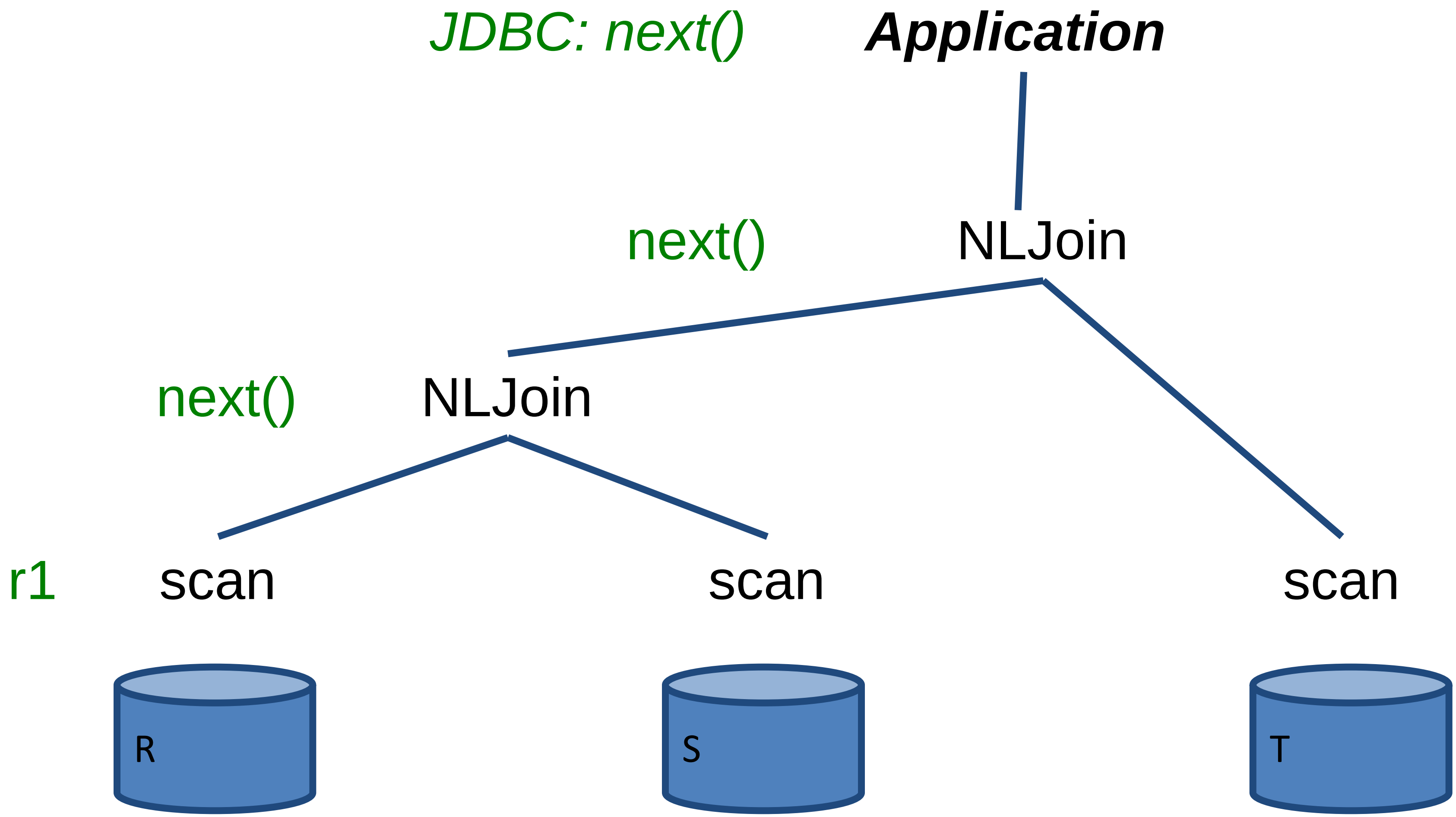
Iterator Model at Work



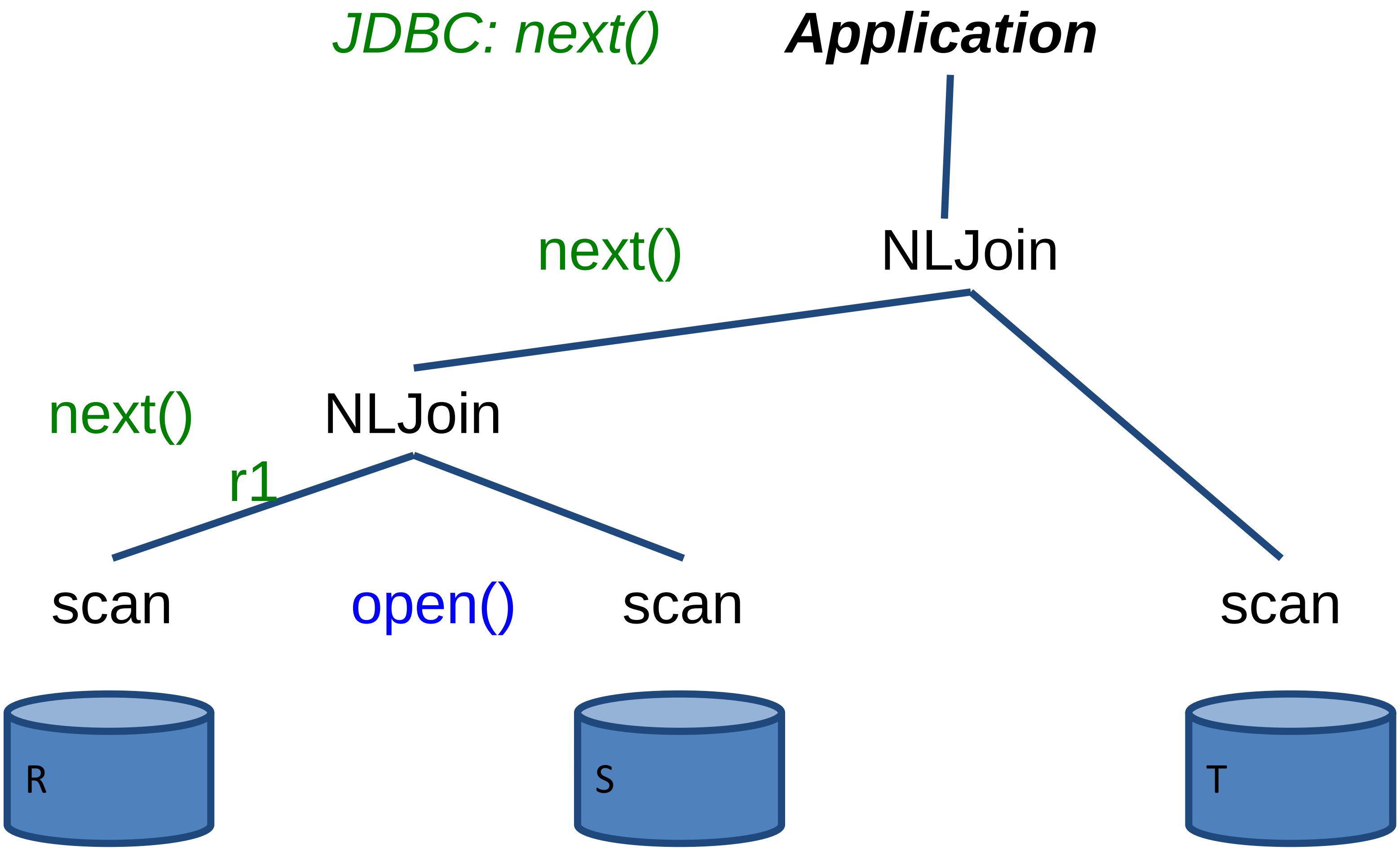
Iterator Model at Work



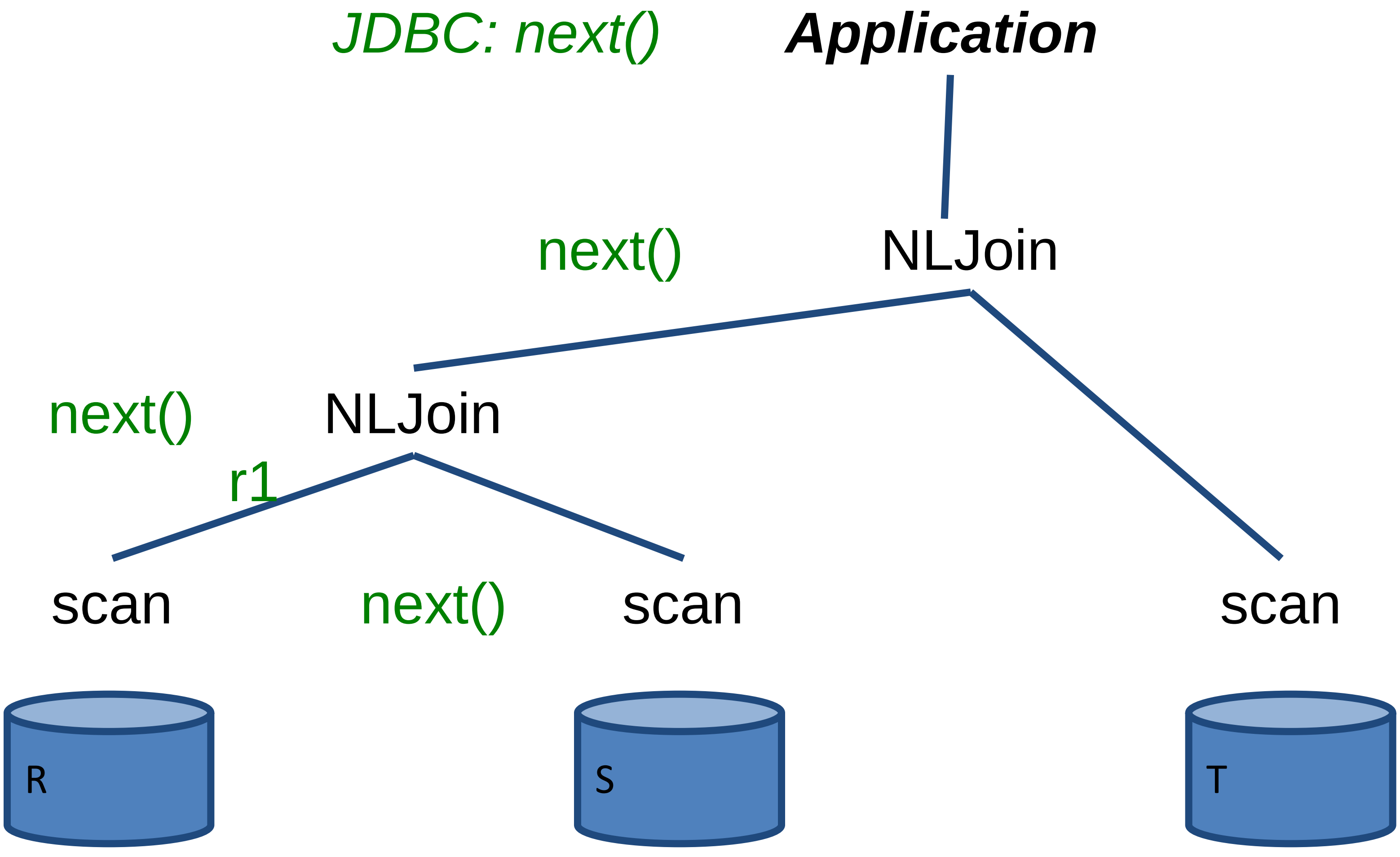
Iterator Model at Work



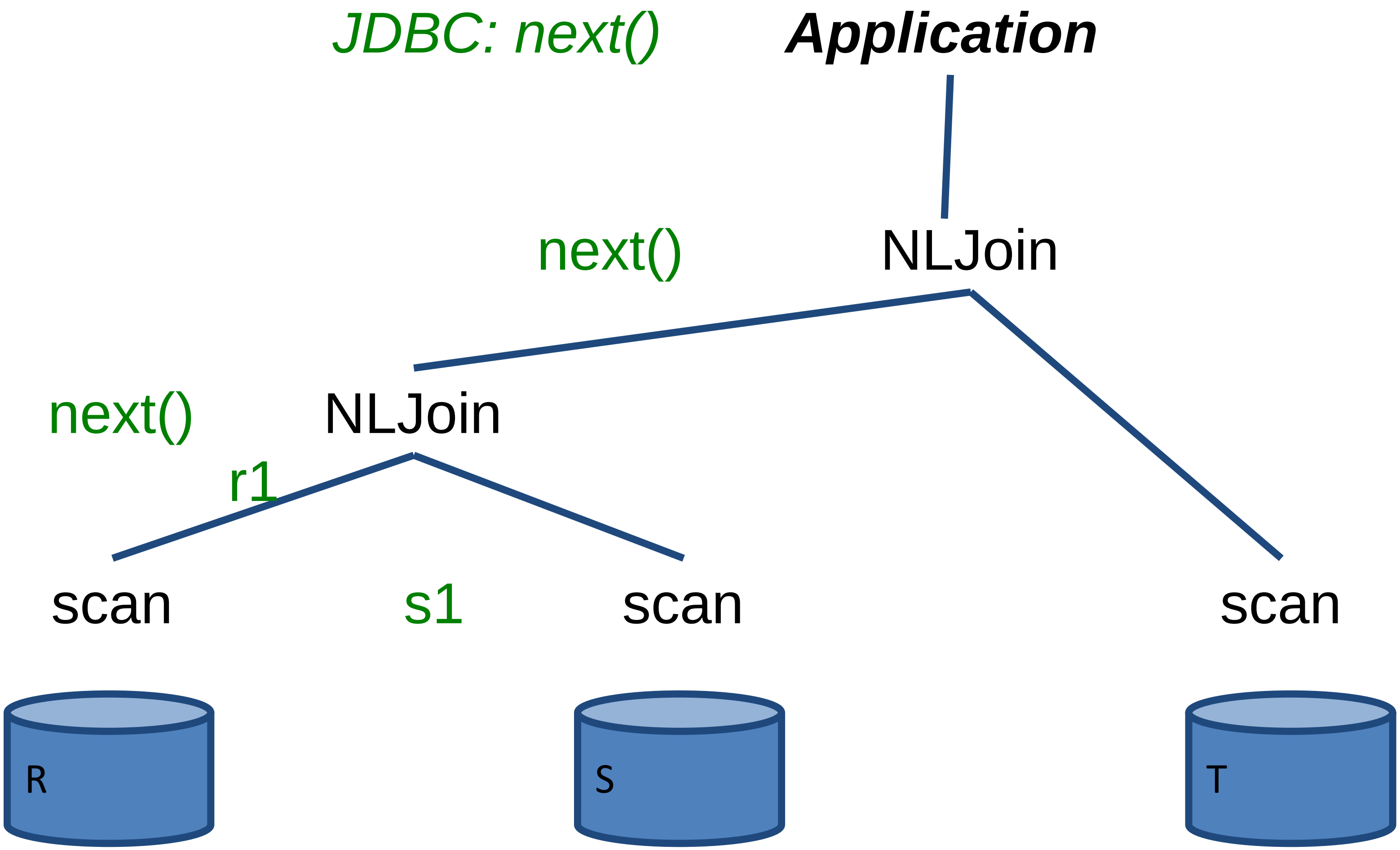
Iterator Model at Work



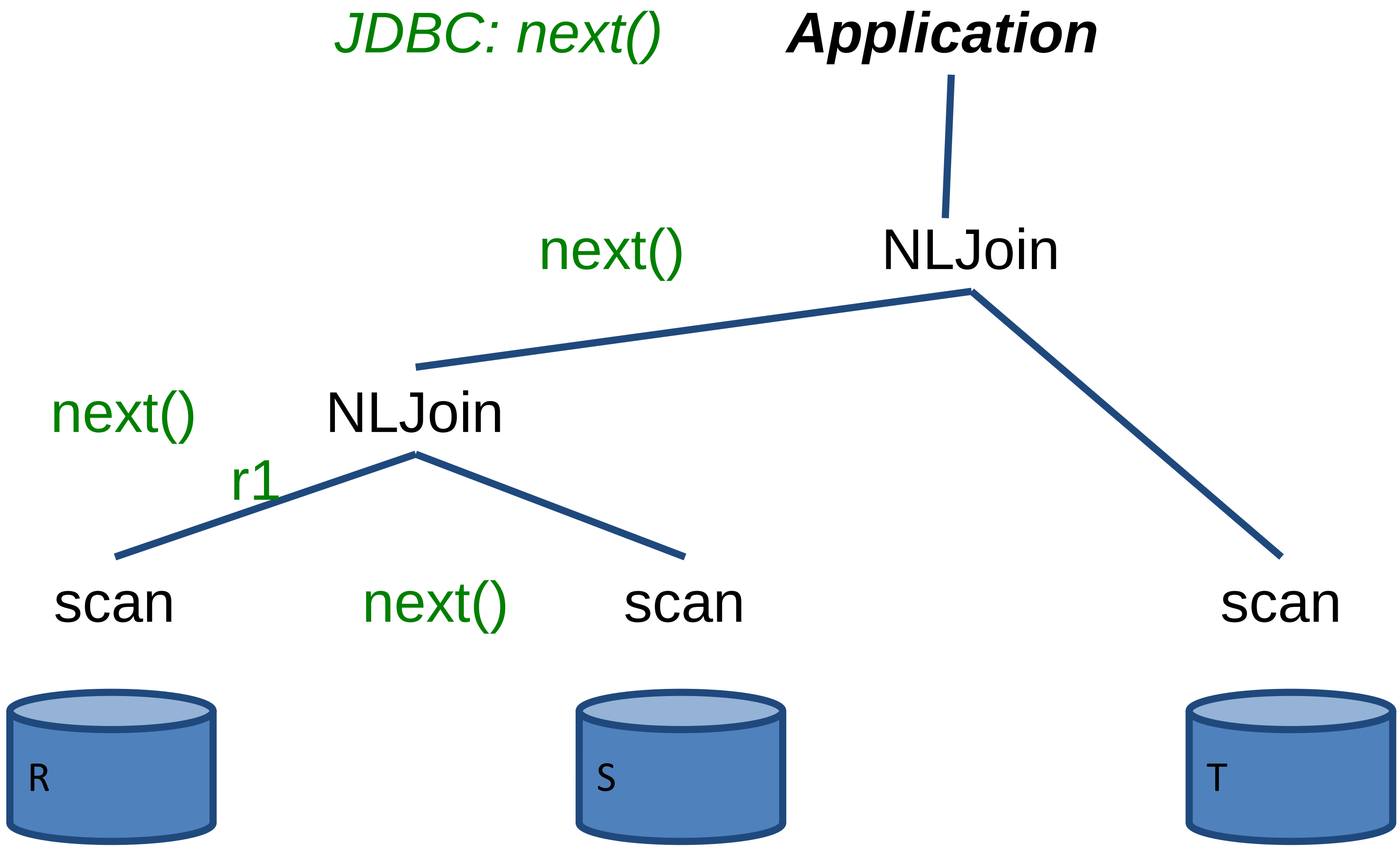
Iterator Model at Work



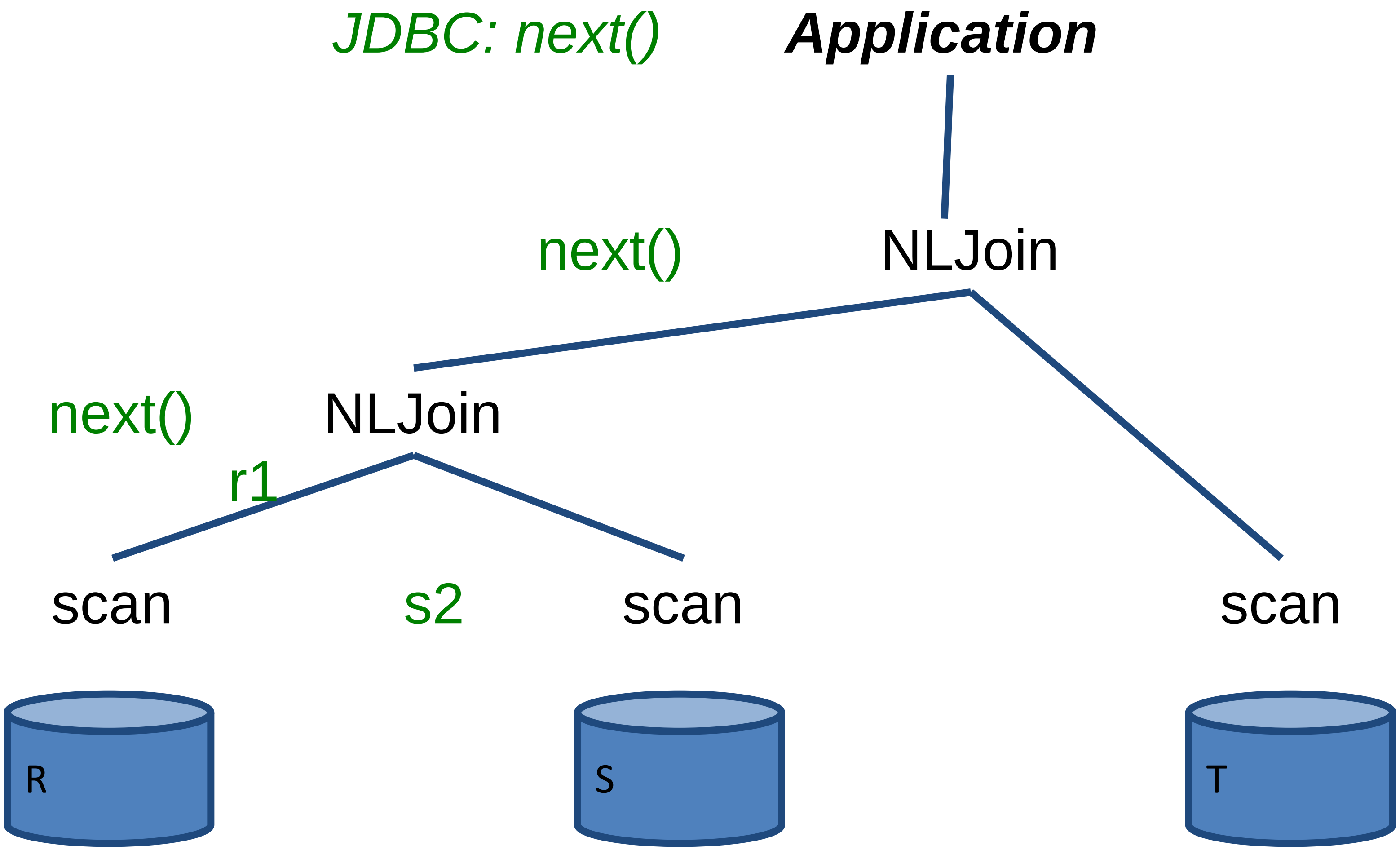
Iterator Model at Work



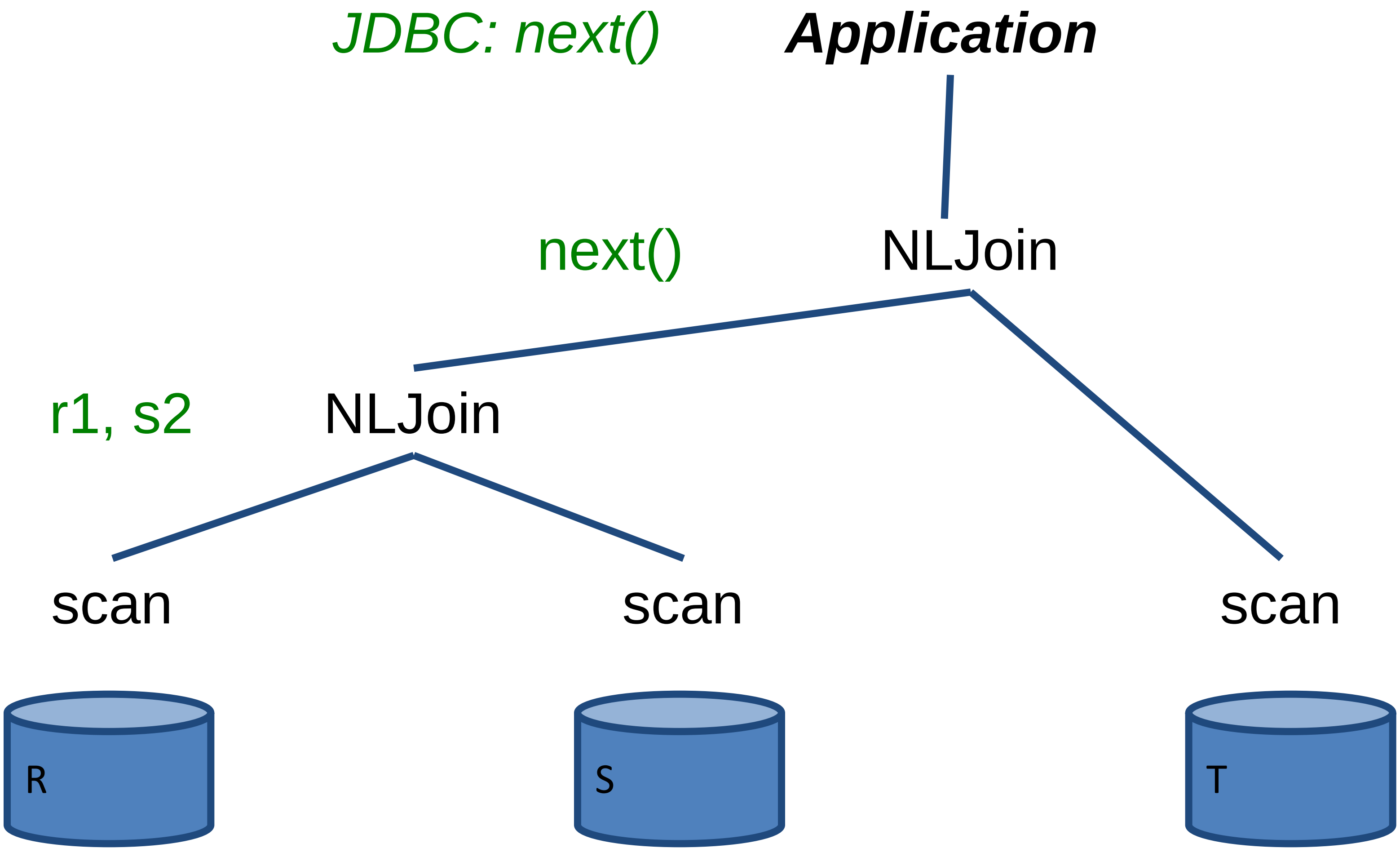
Iterator Model at Work



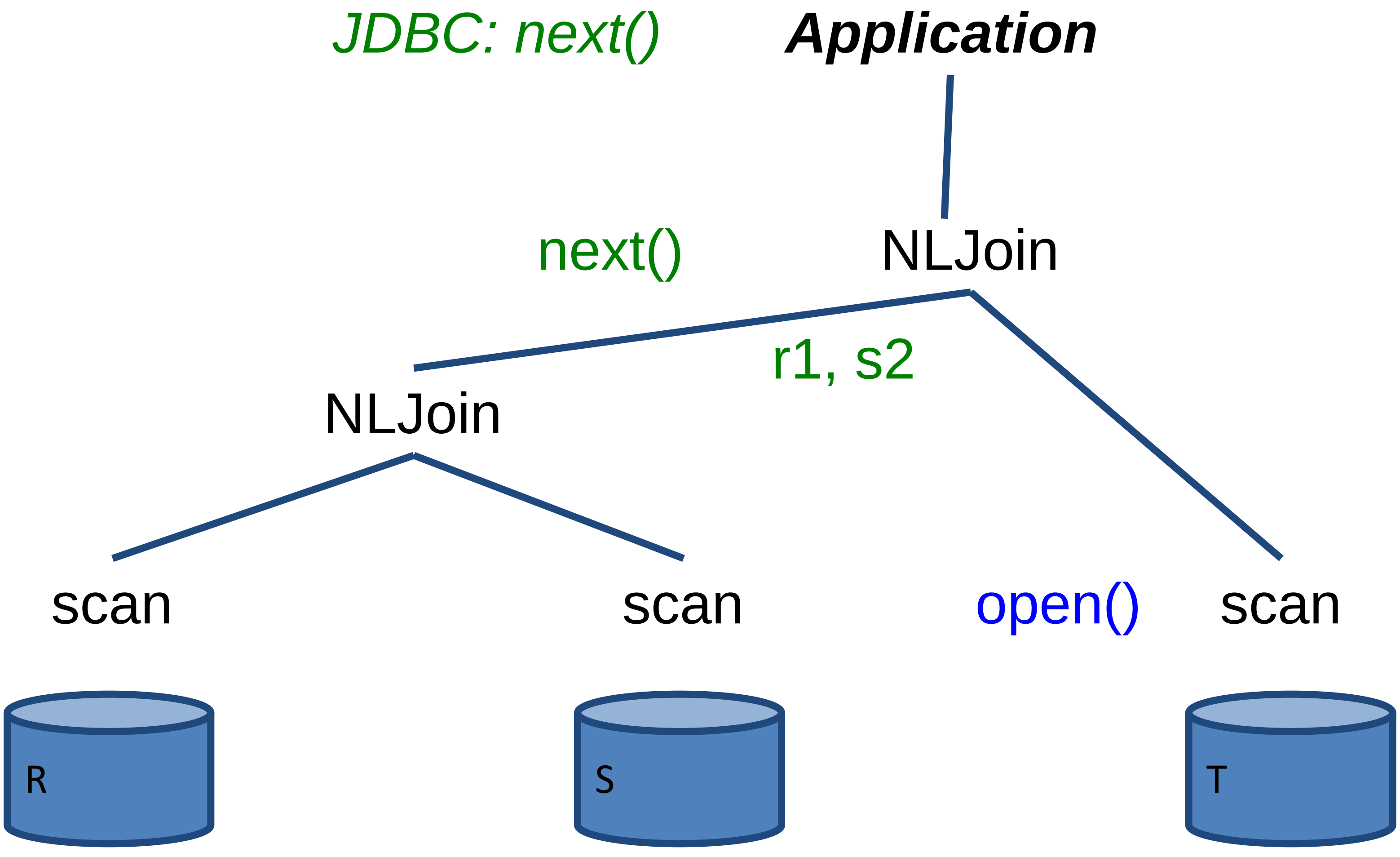
Iterator Model at Work



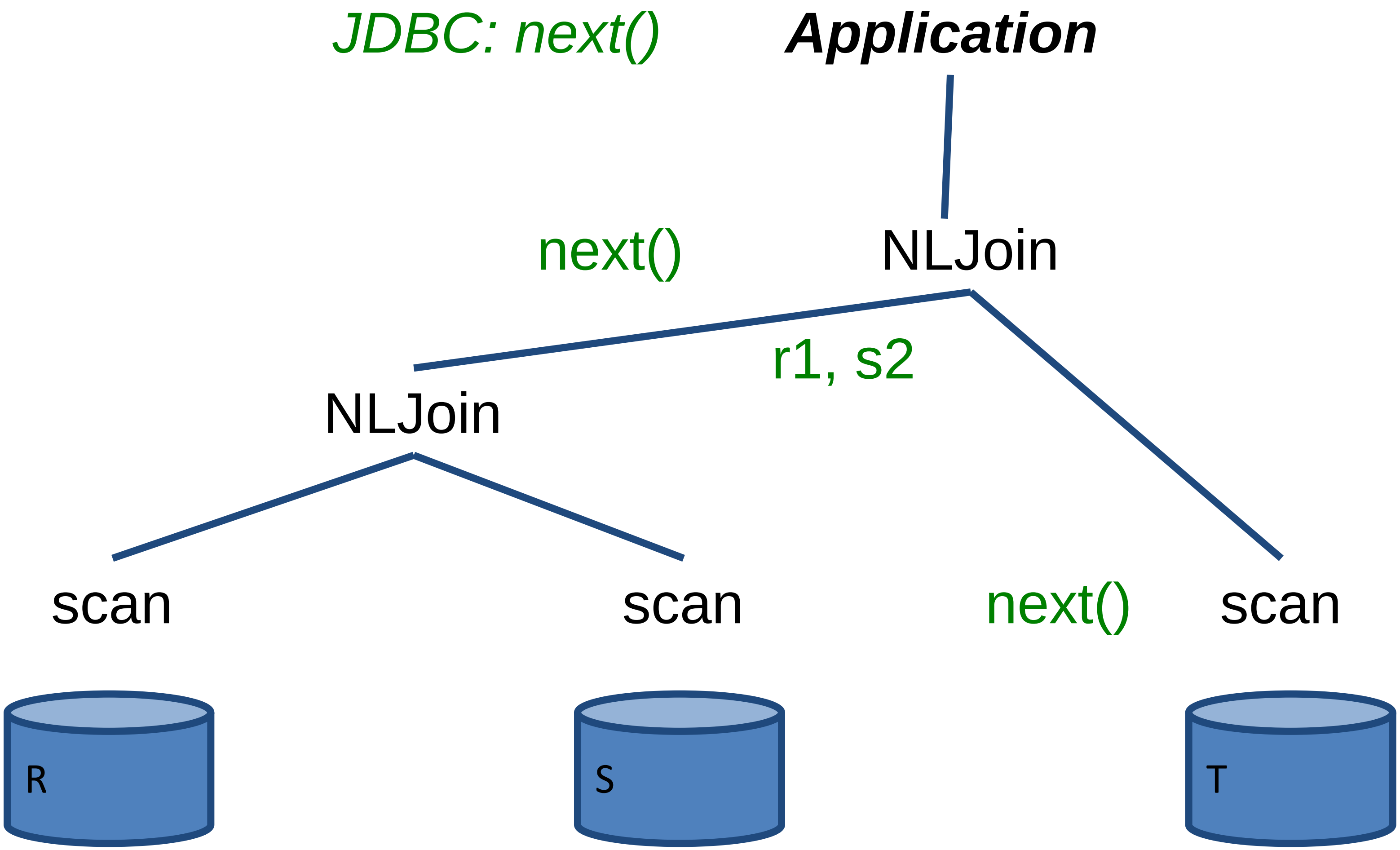
Iterator Model at Work



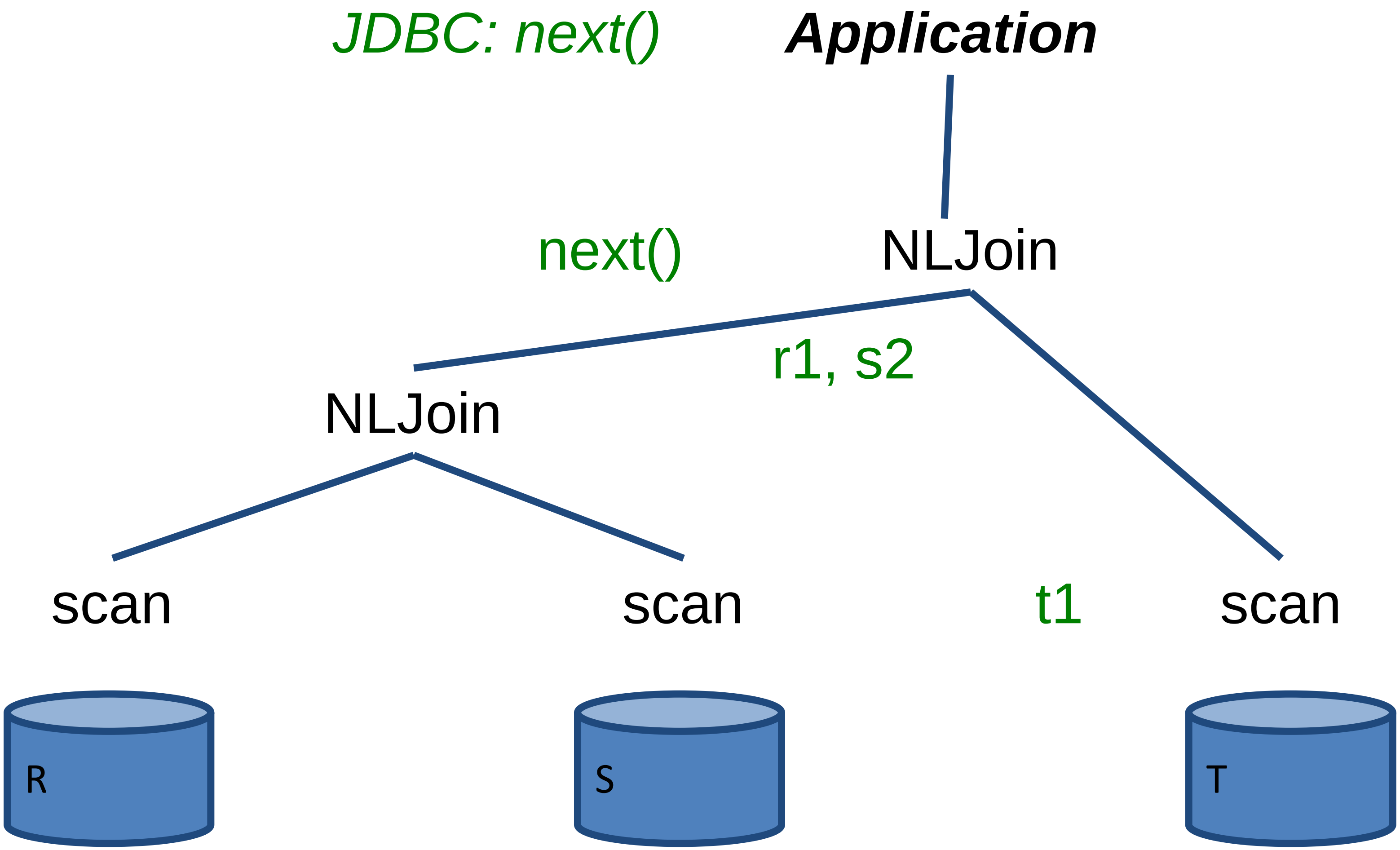
Iterator Model at Work



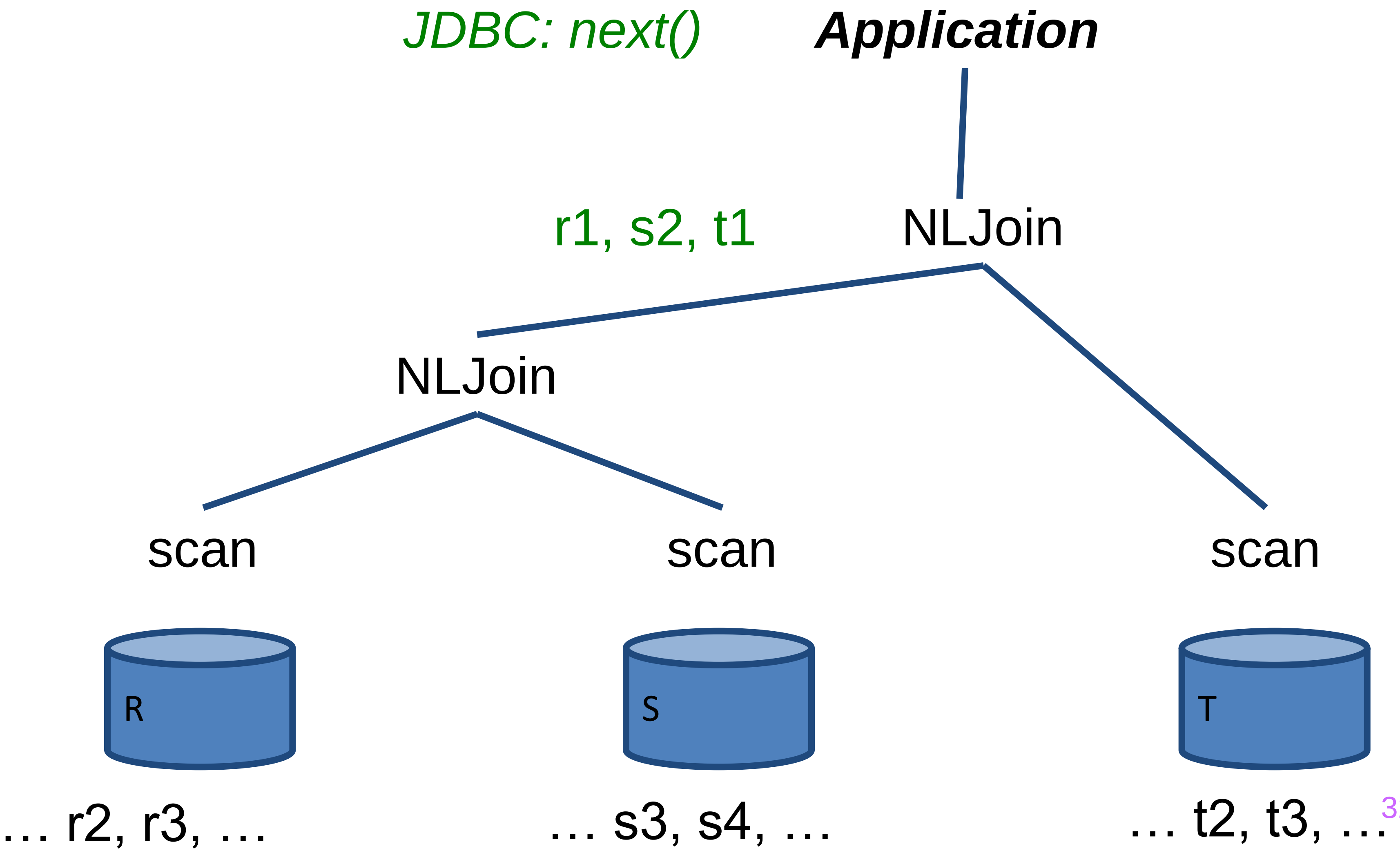
Iterator Model at Work



Iterator Model at Work



Iterator Model at Work



Iterators: Easy & Costly

- Principle
  - data flows bottom up in a plan (i.e. operator tree)
  - control flows top down in a plan
- Advantages
  - generic interface for all operators: great information hiding
  - easy to implement iterators (clear what to do in any phase)
  - works well with JDBC and embedded SQL
  - supports DBmin and other buffer management strategies
  - no overheads in terms of main memory
  - supports pipelining: great if only subset of results consumed
  - supports parallelism and distribution: add special iterators
- Disadvantages
  - high overhead of method calls
  - poor instruction cache locality

Compiler: Optimizations

- Goals:
  1. Reduce the *level of abstraction*
  2. Reduce the *execution cost*
- Concepts
  – Code representation (e.g., algebras)
  – Code transformations (e.g., rules)
  – Cost transformation policy (e.g., enumeration)
  – Code cost estimation

SQL -> Relational Algebra

SQL

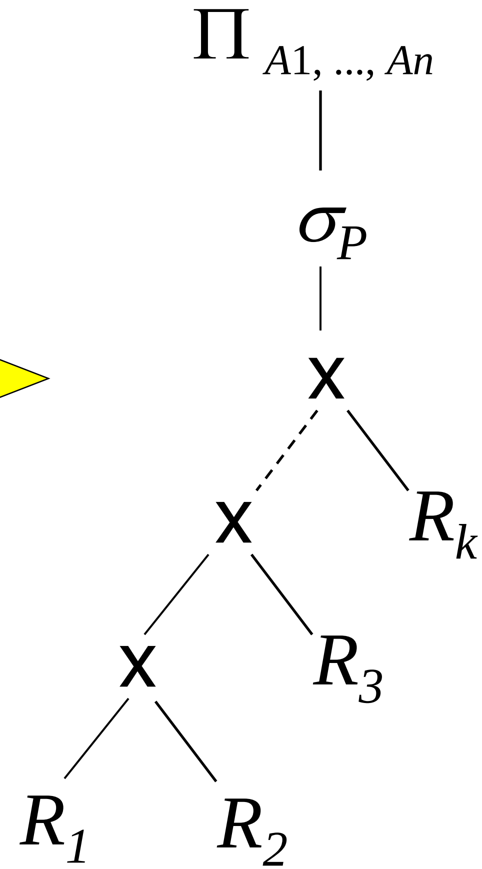
Relational Algebra

$$\Pi_{A1,\ \ldots,\ An}(\sigma_P\ (R_1 \text{ x } \ldots \text{ x } R_k))$$

select $A_1, \ldots, A_n$

from $R_1, \ldots, R_k$

where P;



SQL -> QGM

SQL

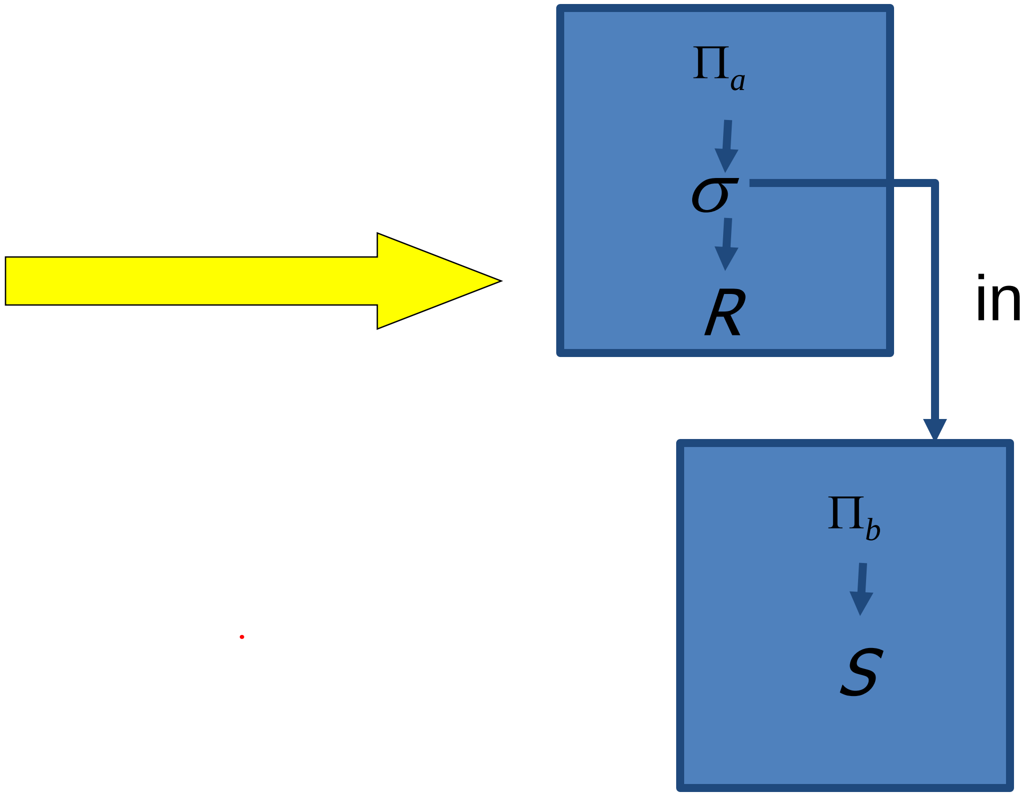
select *a*

from *R*

where *a in (***select** *b*

from *S);*

QGM



Parser

- Generates rel. alg. tree for each sub-query
  - constructs graph of trees: Query Graph Model nodes are subqueries
  - edges represent relationships between subqueries
- Extended rel. algebra because SQL more than RA
  - GROUP BY: Γ operator
  - ORDER BY: sort operator
  - DISTINCT: can be implemented with Γ operator

SQL -> Relational Algebra

SQL

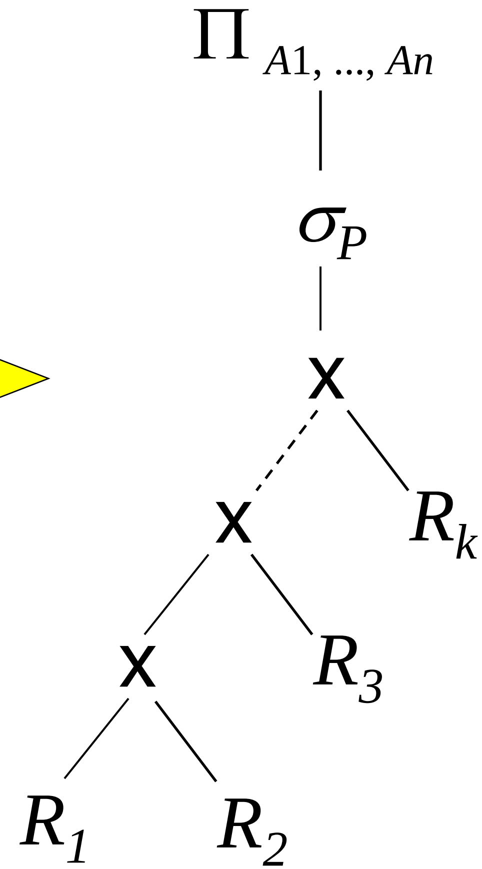
Relational Algebra

$$\Pi_{A1, \ldots, An}(\sigma_P (R_1 \times \ldots \times R_k))$$

select $A_1, \ldots, A_n$

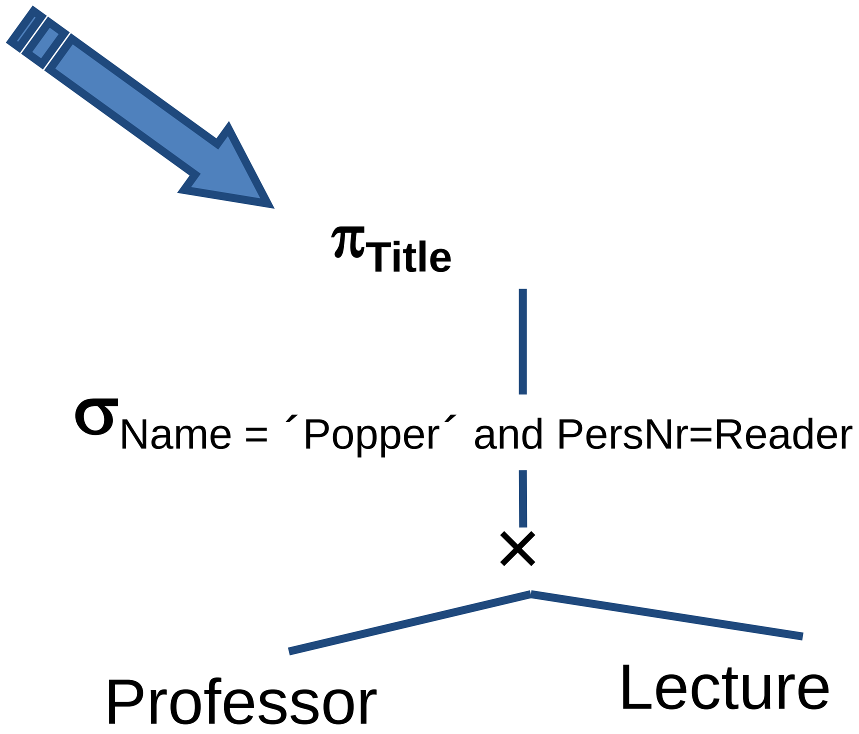
from $R_1, \ldots, R_k$

where P;



Example: SQL -> Relational Algebra

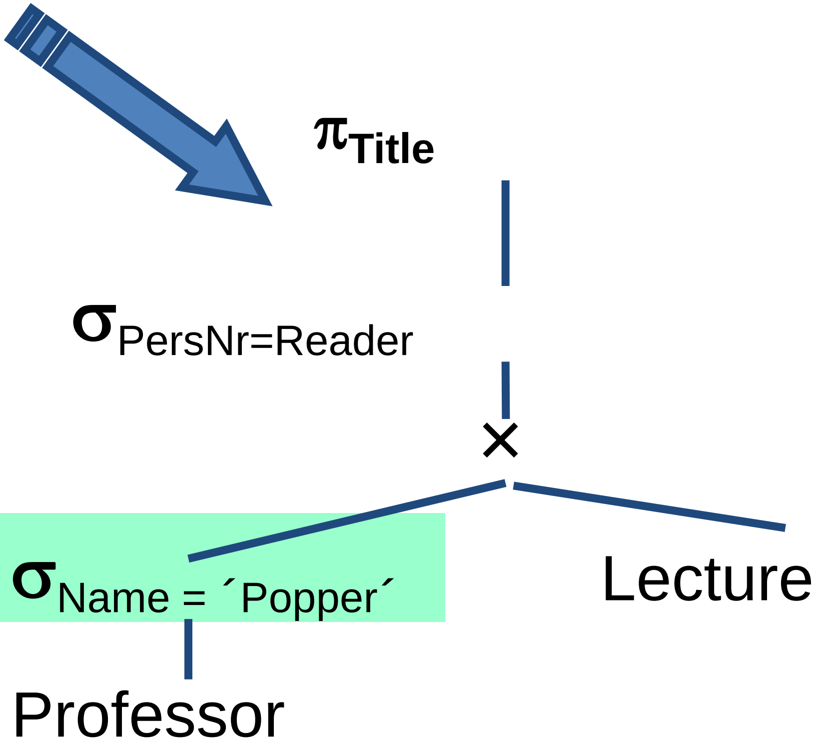
select Title
from Professor, Lecture
where Name = ´Popper´ **and**
PersNr = Reader



$\pi_{\text{Title}} (\sigma_{\text{Name = ´Popper´ and PersNr=Reader}}$ (Professor × Lecture))

First Optimization: Push-down σ

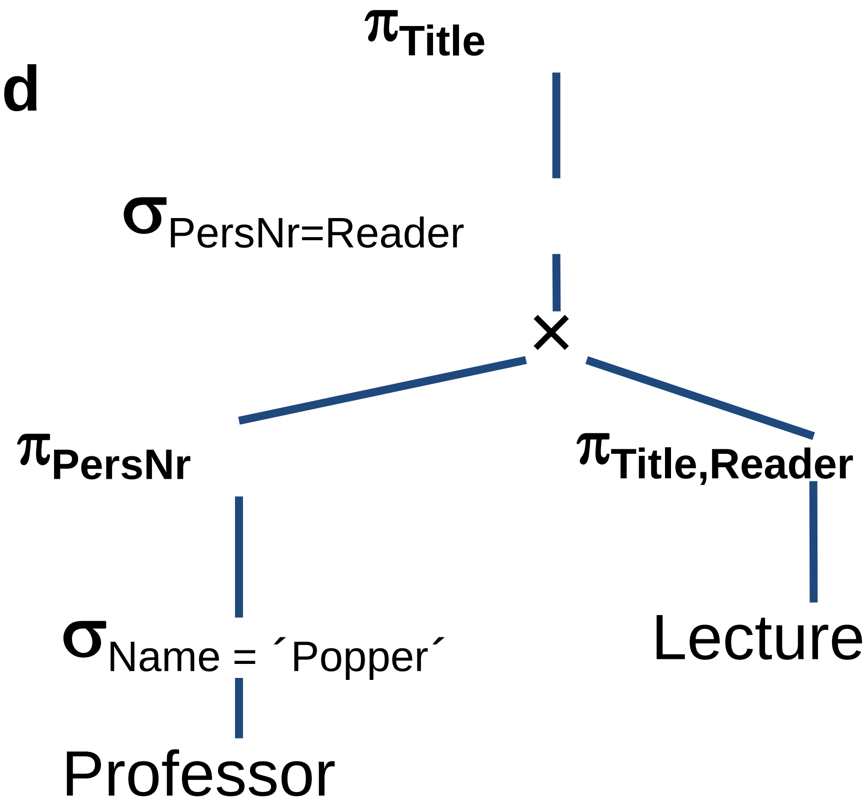
select Title
from Professor, Lecture
where Name = ´Popper´ **and**
PersNr = Reader



$\pi_{\textbf{Title}}$ ($\sigma_{\text{PersNr=Reader}}$ (($\sigma_{\text{Name = ´Popper´}}$ Professor) × Lecture))

Push-down π

select Title
from Professor, Lecture
where Name = ´Popper´ **and**
PersNr = Reader



Correctness: Push-down π

- $\pi_{\text{Title}} (\sigma_{\text{PersNr=Reader}} ((\sigma_{\text{Name = ´Popper´}} \text{Professor}) \times \text{Lecture}))$

  (composition of projections)

- $\pi_{\text{Title}} (\pi_{\text{Title,PersNr,Reader}} (\sigma_{\ldots} ((\sigma_{\ldots} \text{Professor}) \times \text{Lecture})))$

  (commutativity of π and σ)

- $\pi_{\text{Title}} (\sigma_{\ldots} (\pi_{\text{Title,PersNr,Reader}} ((\sigma_{\ldots} \text{Professor}) \times \text{Lecture})))$

  (commutativity of π and σ)

- $\pi_{\text{Title}} (\sigma_{\ldots} (\pi_{\text{PersNr}} (\sigma_{\ldots} \text{Professor}) \times \pi_{\text{Title,Reader}} (\text{Lecture})))$

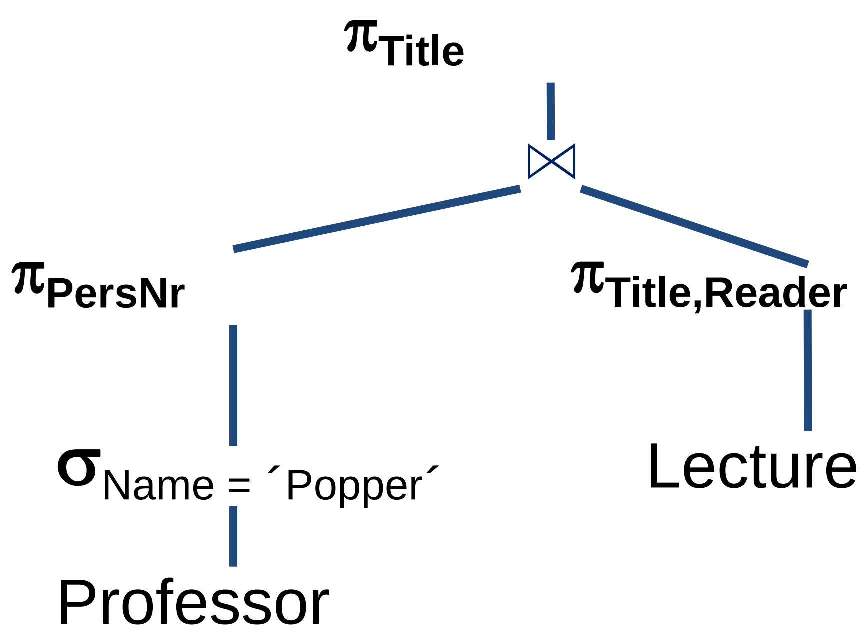
Push down π

- Correctness (see previous slide – example generalizes)

- Why is it good? (almost same reason as for σ)
  - reduces size of intermediate results
  - but: only makes sense if results are materialized; e.g. sort
    - does not make sense if pointers are passed around in iterators

Third Optimization: σ + x = $\bowtie$

select Title
from Professor, Lecture
where Name = ´Popper´ **and**
PersNr = Reader



Third Optimization: σ + x = $\bowtie$

- Correctness by definition of $\bowtie$ operator
- Why is this good?
  - x always done using nested-loops algorithm
    - $\bowtie$ can also be carried out using hashing, sorting, index support
    - choice of better algorithm may result in huge wins
  - x produces large intermediate results
    - results in a huge number of „next()" calls in iterator model
    - method calls are expensive
- Selection, projection push-down are no-brainers
  - make sense whenever applicable
  - do not need a cost model to decide how to apply them
  - (exception: expensive selections, projections with UDF)
  - done in a phase called query rewrite, based on rules
- More complex query rewrite rules…

Unnesting of Views

- Example: Unnesting of Views

```
select A.x
from   A
where y in
   (select y from B)
```

→

```
select A.x
from   A, B
where A.y = B.y
```

- Example: Unnesting of Views

```
select A.x
from   A
where exists
   (select * from B where A.y = B-y)
```

→

```
select A.x
from   A, B
where A.y = B.y
```

- Is this correct? Why is this better?
  - (not trivial at all!!!)

Query Rewrite

- Example: Predicate Augmentation

```
select *
from   A, B, C
where  A.x = B.x
 and B.x = C.x
```

→

```
select *
from   A, B, C
where  A.x = B.x
 and B.x = C.x
 and A.x = C.x
```

Why is that useful?

Pred. Augmentation: Why good?

A (odd numbers)

...	x
...	1
...	3
...	5
...	...

B (all numbers)

...	x
...	1
...	2
...	3
...	...

C (even numbers)

...	x
...	2
...	4
...	6
...	...

- Cost((A ⋈ C) ⋈ B) < Cost((A ⋈ B) ⋈ C)
  - get second join for free
- Query Rewrite does not know that, …
  - but it knows that it might happen and hopes for optimizer…
- Codegen gets rid of unnecessary predicates (e.g., A.x = B.x)

Query Optimization

- Two tasks
  - Determine order of operators
  - Determine algorithm for each operator (hash vs sort)
- Components of a query optimizer
  - Search space
  - Cost model
  - Enumeration algorithm
- Working principle
  - Enumerate alternative plans
  - Apply cost model to alternative plans
  - Select plan with lowest expected cost

Query Opt.: Does it matter?

- A x B x C
  - size(A) = 10,000
  - size(B) = 100
  - size(C) = 1
  - cost(X x Y) = size(X) + size(Y)

- cost((A x B) x C) = **1,010,001**
  - cost(A x B) = 10,100
  - cost(X x C) = 1,000,001 with X = A x B

- cost (A x (B x C)) = **10,201**
  - cost(B x C) = 101
  - cost(A x X) = 10,100 with X = B x C

Query Opt.: Does it matter?

- A x B x C
  - size(A) = 1000
  - size(B) = 1
  - size(C) = 1
  - **cost(X x Y) = size(X) * size(Y)**

- cost((A x B) x C) = **2000**
  - cost(A x B) = 1000
  - cost(X x C) = 1000 with X = A x B

- cost (A x (B x C)) = **1001**
  - cost(B x C) = 1
  - cost(A x X) = 1000 with X = B x C

Search Space: Rel. Algebra

- Associativity of joins:
  $(A \bowtie B) \bowtie C = A \bowtie (B \bowtie C)$

- Commutativity of joins:
  $A \bowtie B = B \bowtie A$

- Many more rules
  - see Kemper/Eickler or Garcia-Molina text books

- What is better: $A \bowtie B$ or $B \bowtie A$?
  - it depends
  - need cost model to make decision

Search Space: Group Bys

SELECT … FROM R, S WHERE R.a = S.a GROUP BY R.a, S.b;

- $\Gamma_{R.a,\ S.b}(R \bowtie S)$

- $\Gamma_{S.b}(\Gamma_{R.a}(R) \bowtie S)$

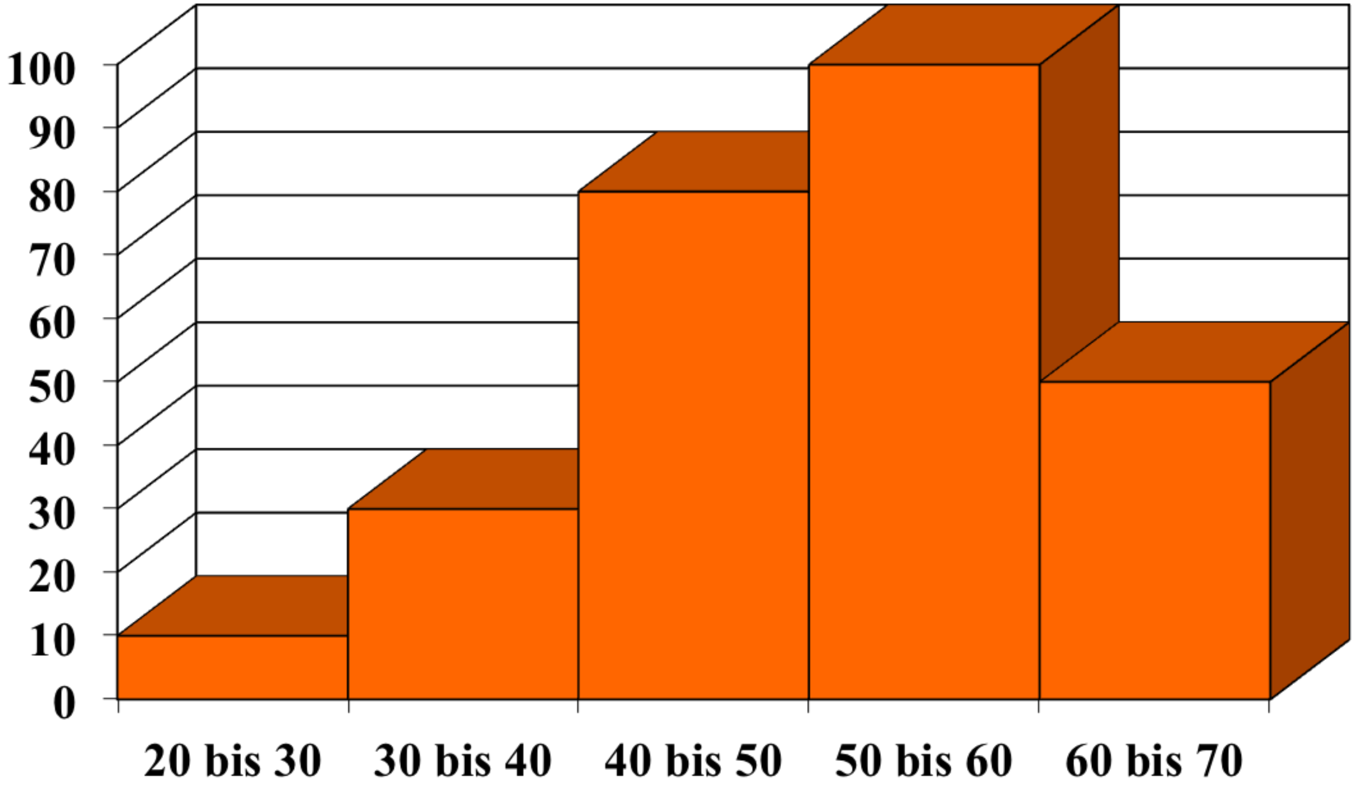
- Often, many possible ways to split & move group-bys
  - again, need cost model to make right decisions

Cost Model

- Cost Metrics
  - Response Time (consider parallelism)
  - Resource Consumption: CPU, IO, network
  - $ (often equivalent to resource consumption)
- Principle
  - Understand algorithm used by each operator (sort, hash, …)
    - estimate available main memory buffers
    - estimate the size of inputs, intermediate results
  - Combine cost of operators:
    - sum for resource consumption
    - max for response time (but keep track of bottlenecks)
- Uncertainties
  - estimates of buffers, interference with other operators
  - estimates of intermediate result size (histograms)

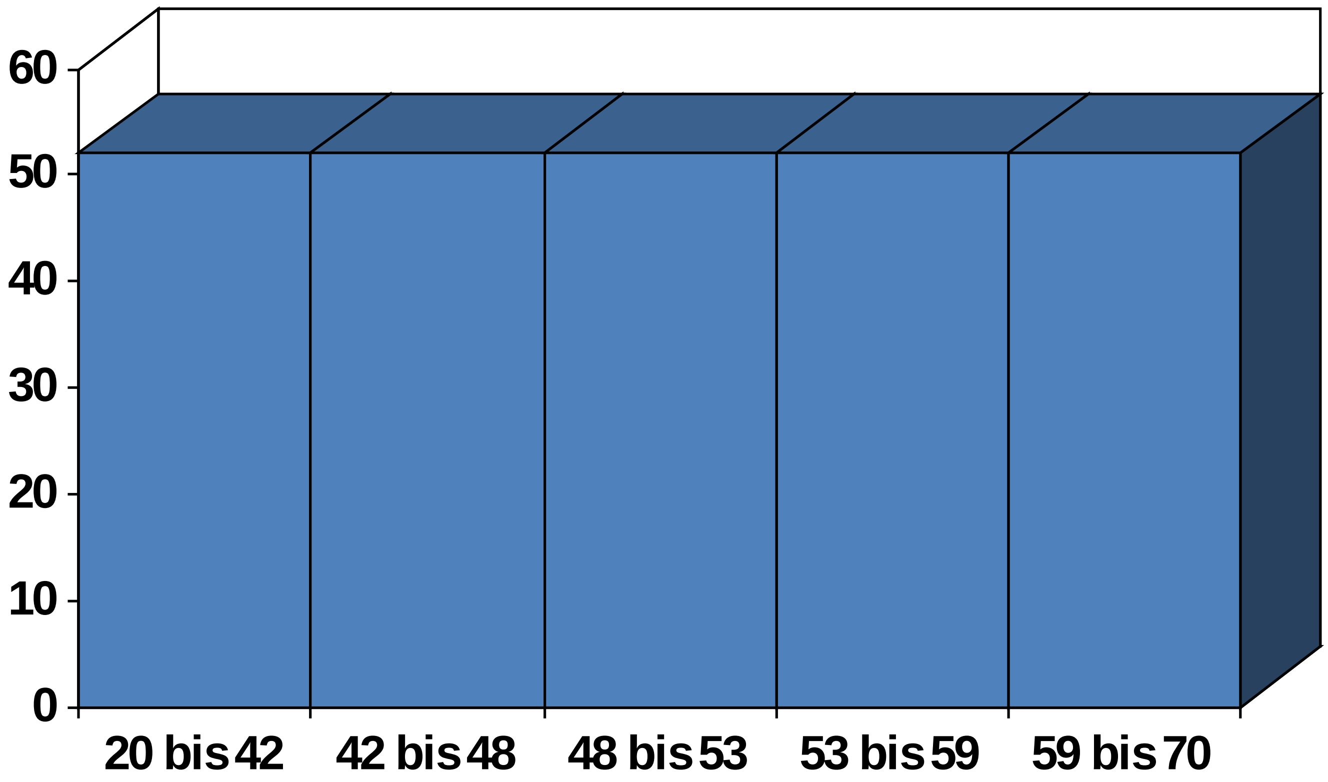
Equi-Width Histogram

SELECT * FROM person WHERE 25 < age < 40;



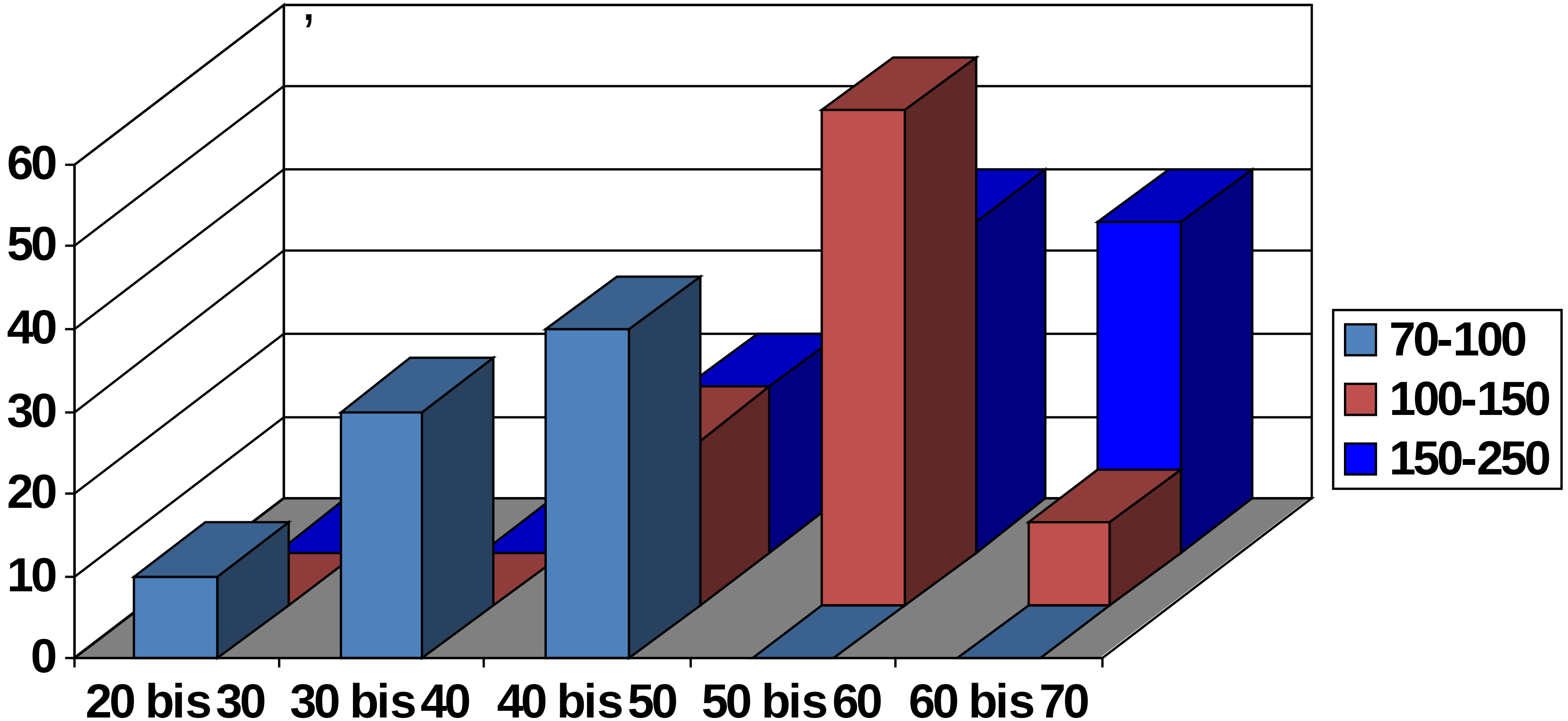
Equi-Depth Histogram

SELECT * FROM person WHERE 25 < age < 40;



Multi-Dimensional Histogram

SELECT * FROM person
WHERE 25 < age < 40 AND salary > 200;
;



Enumeration Algorithms

- Query Optimization is NP hard
  - even ordering or Cartesian products is NP hard
  - in general impossible to predict complexity for given query
- Overview of Algorithms
  - Dynamic Programming (good plans, exp. complexity)
  - Greedy heuristics (e.g., highest selectivity join first)
  - Randomized Algorithms (iterative improvement, Sim.An., ...)
  - Other heuristics (e.g., rely on hints by programmer)
  - Smaller search space (e.g., deep plans, limited group-bys)
- Products
  - Dynamic Programming used by many systems
  - Some systems also use greedy heuristics in addition