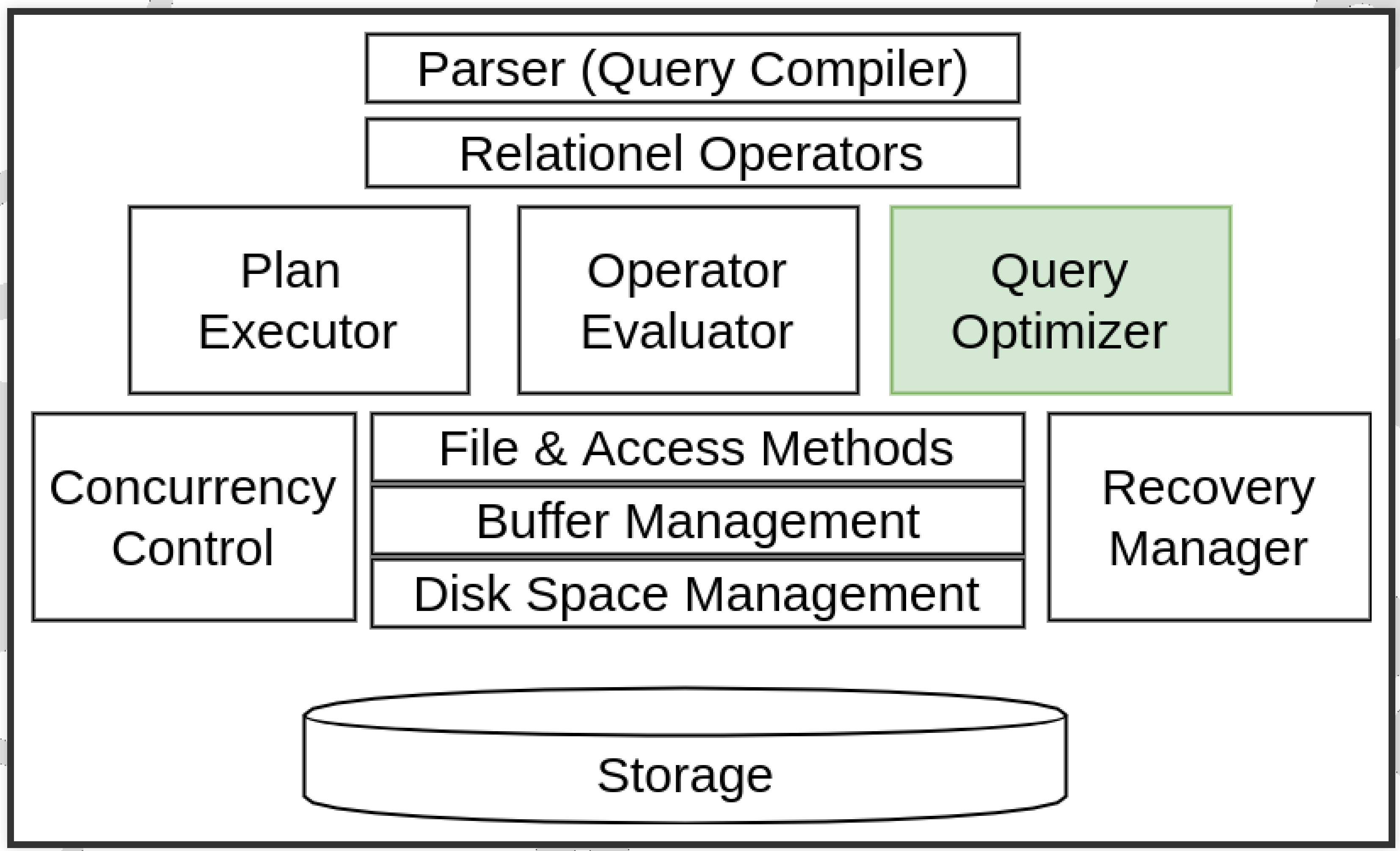
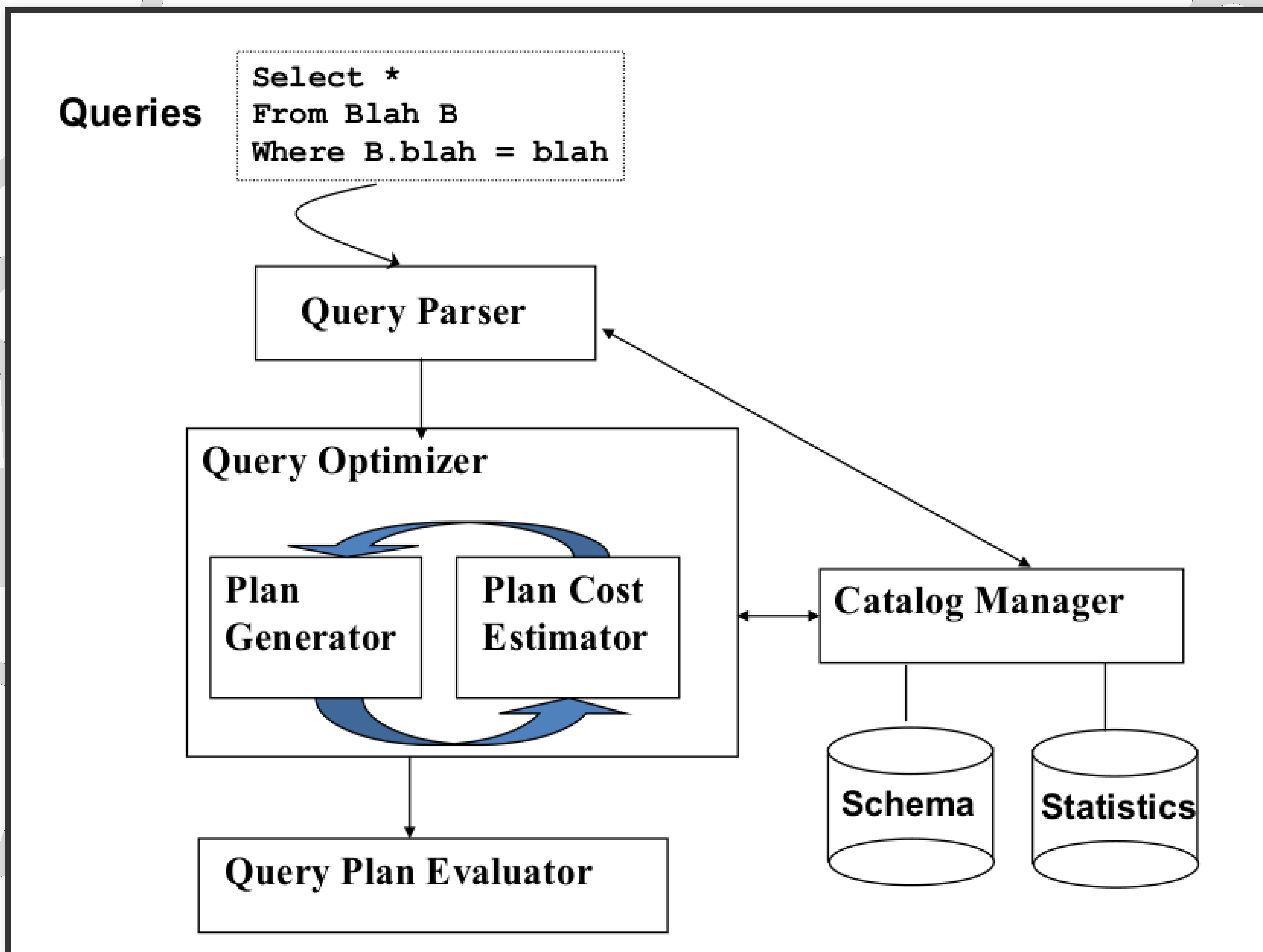
QUERY OPTIMIZATION

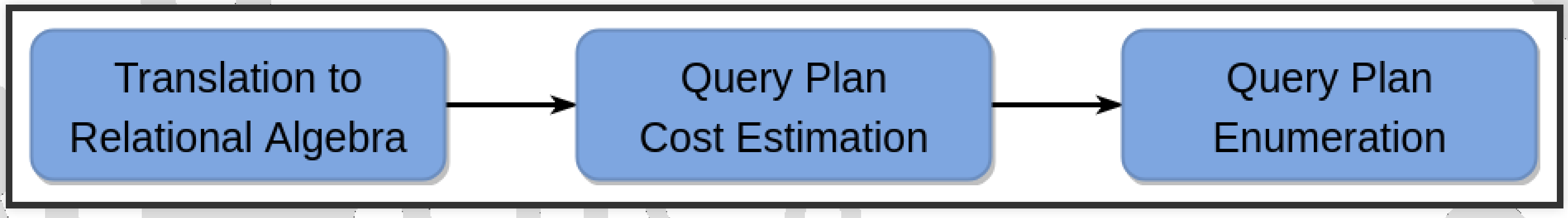
QUERY OPTIMIZATION



QUERY SUB-SYSTEM



ROADMAP


Translation to Relational Algebra
Query Plan Cost Estimation
Query Plan Enumeration

QUERY BLOCKS: UNITS OF OPTIMIZATION

- An SQL query is parsed into a collection of query blocks
  - optimize one block at a time.
- Nested blocks are usually treated as calls to a subroutine
  - called once per outer tuple.
  - This is an over-simplification, wait till we learn more about nested queries.

SQL EXTENDS RELATIONAL ALGEBRA

- SQL is **more** powerful than relational algebra
  - extend relational algebra to include aggregate ops: GROUP BY, HAVING

How is this query block expressed?

```
SELECT S.sname
FROM Sailors S
WHERE S.age IN (constant set from subquery)
```

$$\pi_{sname}(\sigma_{(\text{age in set from subquery})} Sailors)$$

And this query block?

```
SELECT MAX (S2.age)
FROM Sailors S2
GROUP BY S2.rating
```

$$\pi_{Max(age)}(GroupBy_{Rating}(Sailors))$$

TRANSLATING SQL TO RELATIONAL ALGEBRA

```
SELECT S.sid, MIN (R.day)
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = “red”
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

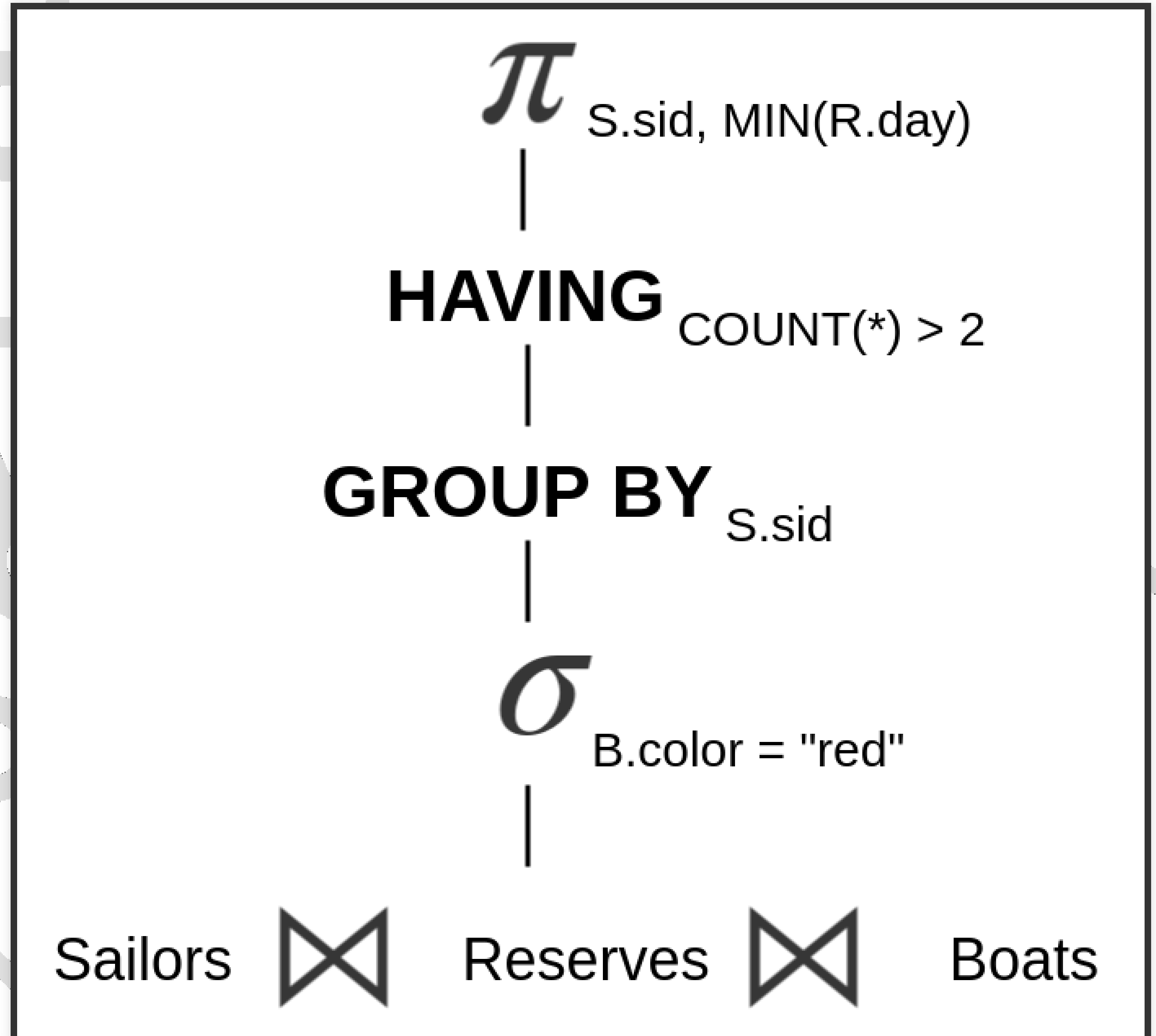
For each sailor with at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

TRANSLATING SQL TO RELATIONAL ALGEBRA

```sql
SELECT S.sid, MIN (R.day)
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = “red”
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

$$\pi_{S.sid, Min(R.day)}(\\ HAVING_{COUNT(*)>2}(\\ GROUPBy_{S.sid}(\sigma_{B.color="red"}(\\ Sailors \bowtie Reserves \bowtie Boats))))$$

TRANSLATING SQL TO RELATIONAL ALGEBRA



RELATIONAL ALGEBRA EQUIVALENCES

Allow us to choose different join orders and to `push' selections and projections ahead of joins.

Selections:

- **Cascade:** $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R)) \ldots)$
- **Commute** $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$

Projections

- **Cascade:** $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{a1,\ldots,an}(R))\ldots)$

RELATIONAL ALGEBRA EQUIVALENCES

Cartesian Product

- **Associative:** $R \times (S \times T) \equiv (R \times S) \times T$
- **Cummutative:** $R \times S \equiv S \times R$

This means we can join in any order (But… beware of the cartesian product!)

Selection between attributes of the two arguments of a cross-product converts cross-product to a join.

MORE EQUIVALENCES

Eager projection

- Rule of thumb: can project out anything not needed by operators above
- Can cascade and “push” some projections thru selection
- Can cascade and “push” some projections below one side of a join

A selection on just attributes of R commutes with $R \bowtie S$. (i.e.,
$\sigma(R \bowtie S) \equiv \sigma(R) \bowtie S$)

ELEMENTS OF QUERY OPTIMIZATION (1)

- **A closed set of operators**
  - Relational ops (table in, table out)
  - Encapsulation based on iterators
- **Plan space**
  - Based on relational equivalences
  - Different algorithms of a operator

ELEMENTS OF QUERY OPTIMIZATION (2)

- **Cost Estimation**, based on
  - Cost formulas
  - Size estimation, based on
    - Catalog information on base tables
    - Selectivity (Reduction Factor) estimation
- **A search algorithm**
  - To sift through the plan space based on cost!

EXAMPLE QUERY SCHEMA

Sailors

Sailors(sid: integer, sname: string, rating: integer, age: real)

- Each tuple is 50 bytes long, 80 tuples per page, 500 pages
- Assume **10** different ratings

Reserves

Reserves(sid:integer, bid:integer, day: date, rname: string)

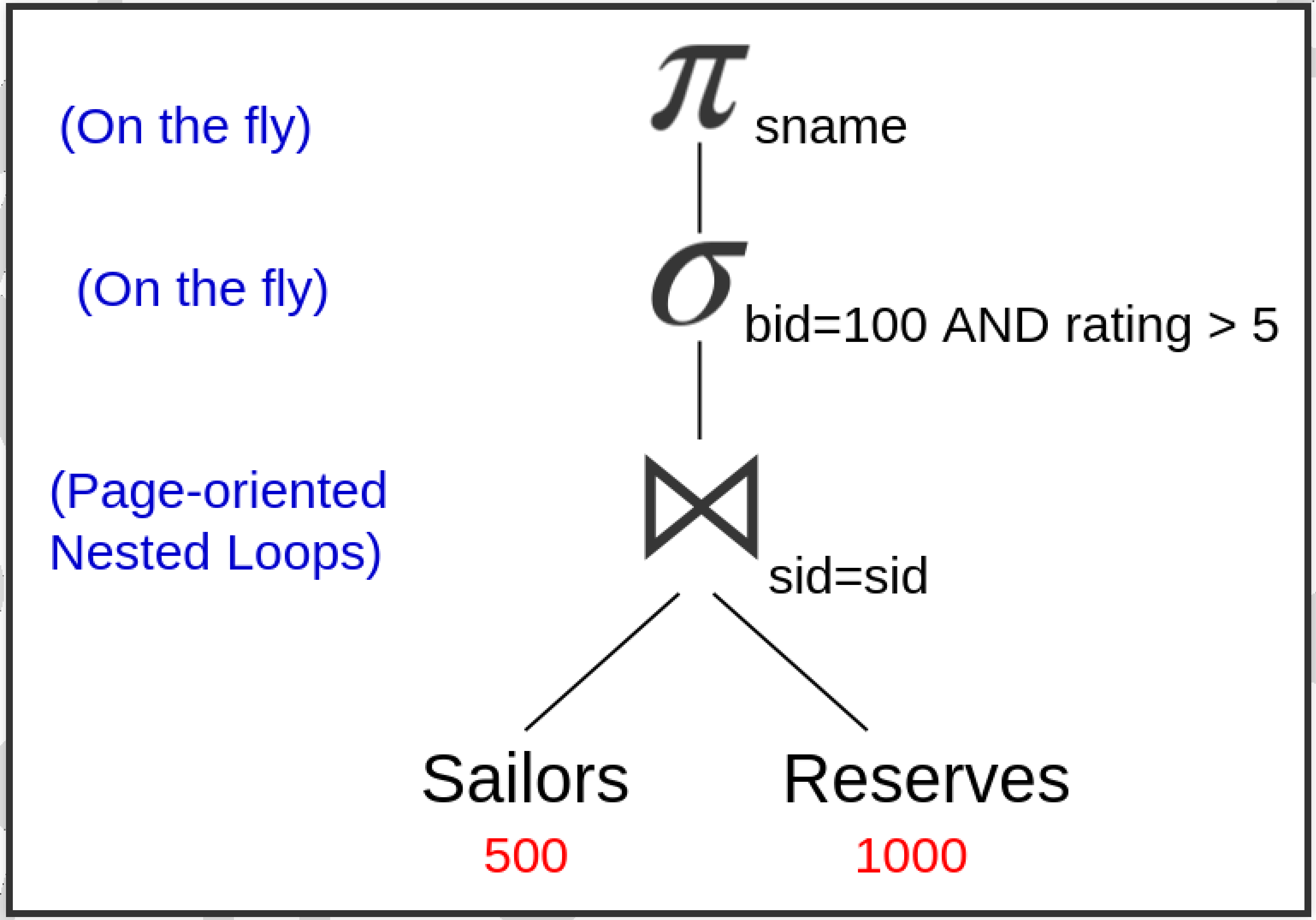
- Each tuple is 40 bytes long, 100 tuples per page, 1000 pages
- Assume **100** boats

Assume we have 5 pages in our buffer pool!

MOTIVATING EXAMPLE

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5
```

MOTIVATING EXAMPLE



MOTIVATING EXAMPLE

Misses several opportunities:

- selections could have been `pushed’ earlier,
- made no use of any available indexes, etc.

Goal of optimization: To find more efficient plans that compute the same answer.

SELECTIVITY CALCULATION

Sailors: 500 pages, 80 tuples per page, 10 ratings

Selectivity of `S.rating > 5`?

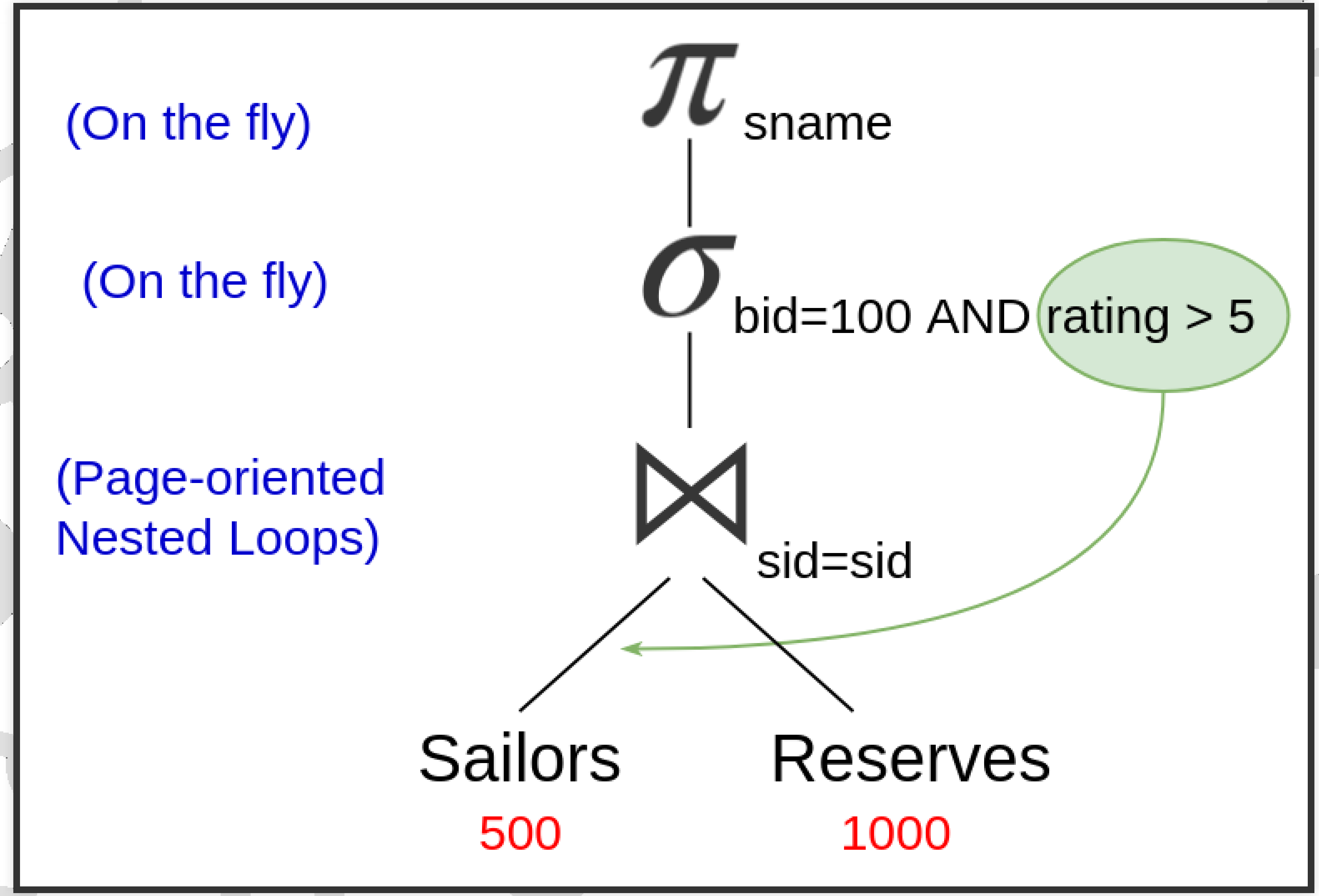
- 1/2 → 500*80/2 = 20,000 tuples
- 20,000/80 = 250 pages

Reserves: 1000 pages, 100 tuples per page, 100 boats

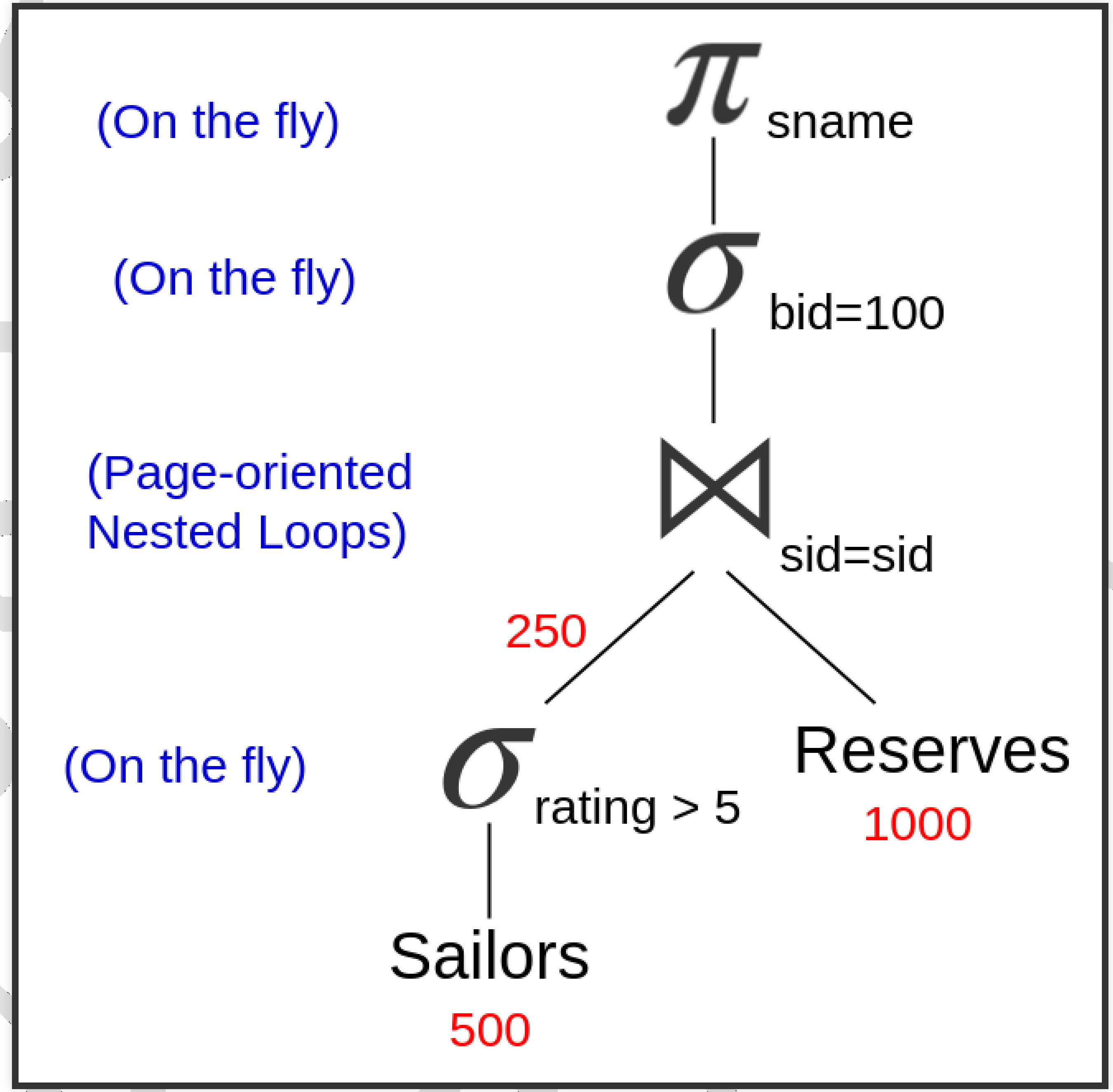
Selectivity of R.bid = 100?

- 1/100 → 1000*100/100 = 1000 tuples
- 1000/100 = 10 pages

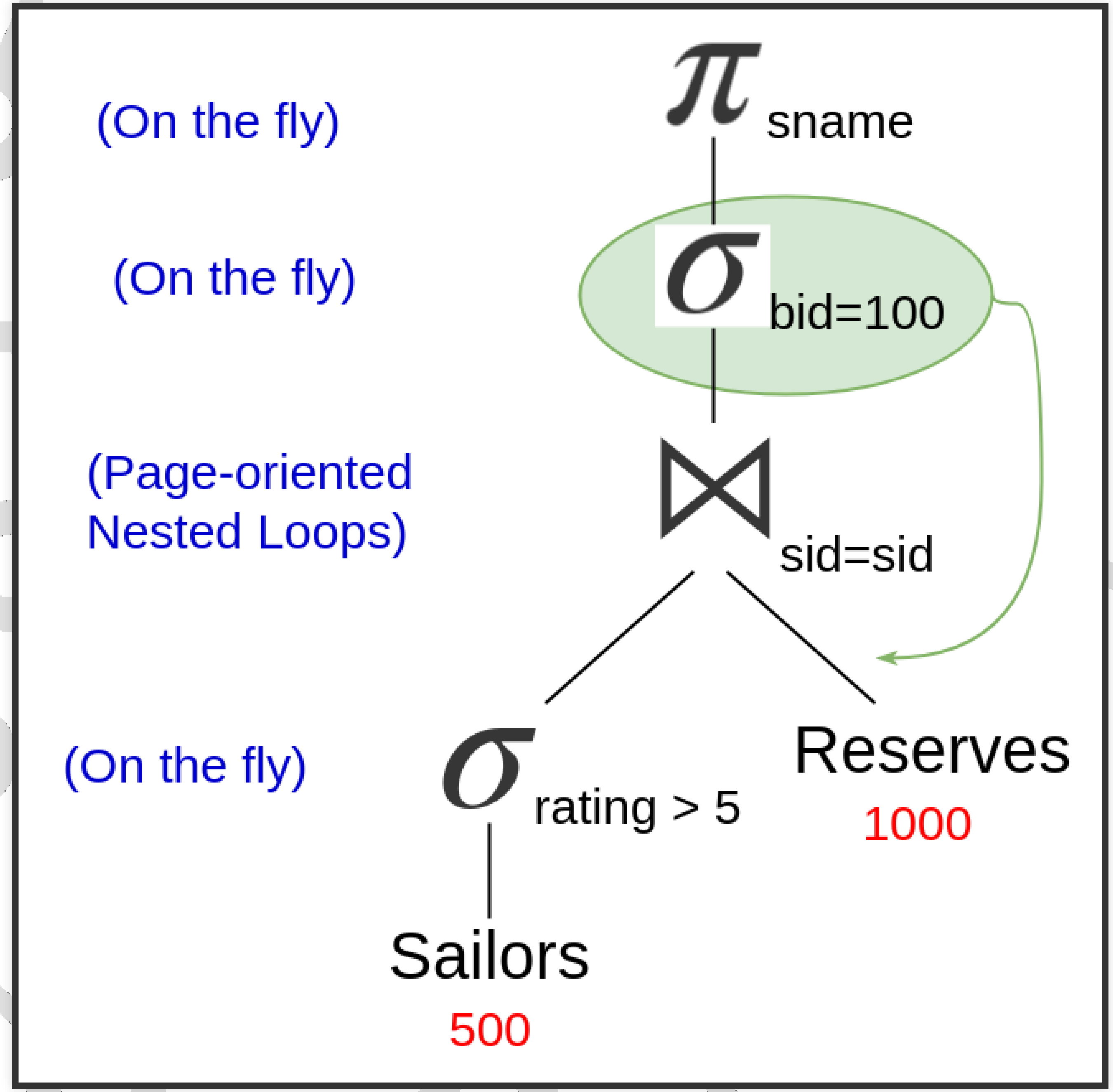
MOTIVATING EXAMPLE



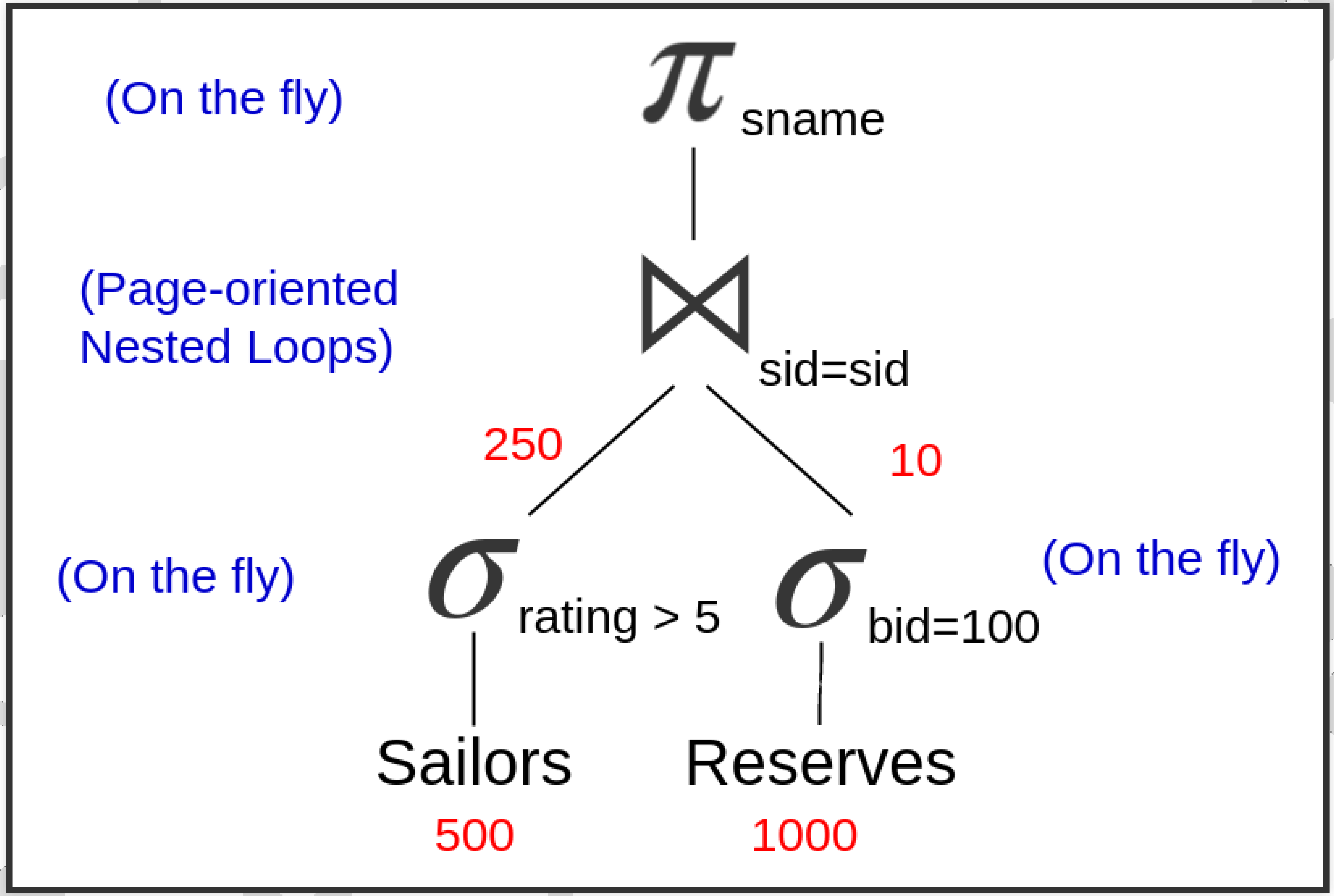
MOTIVATING EXAMPLE



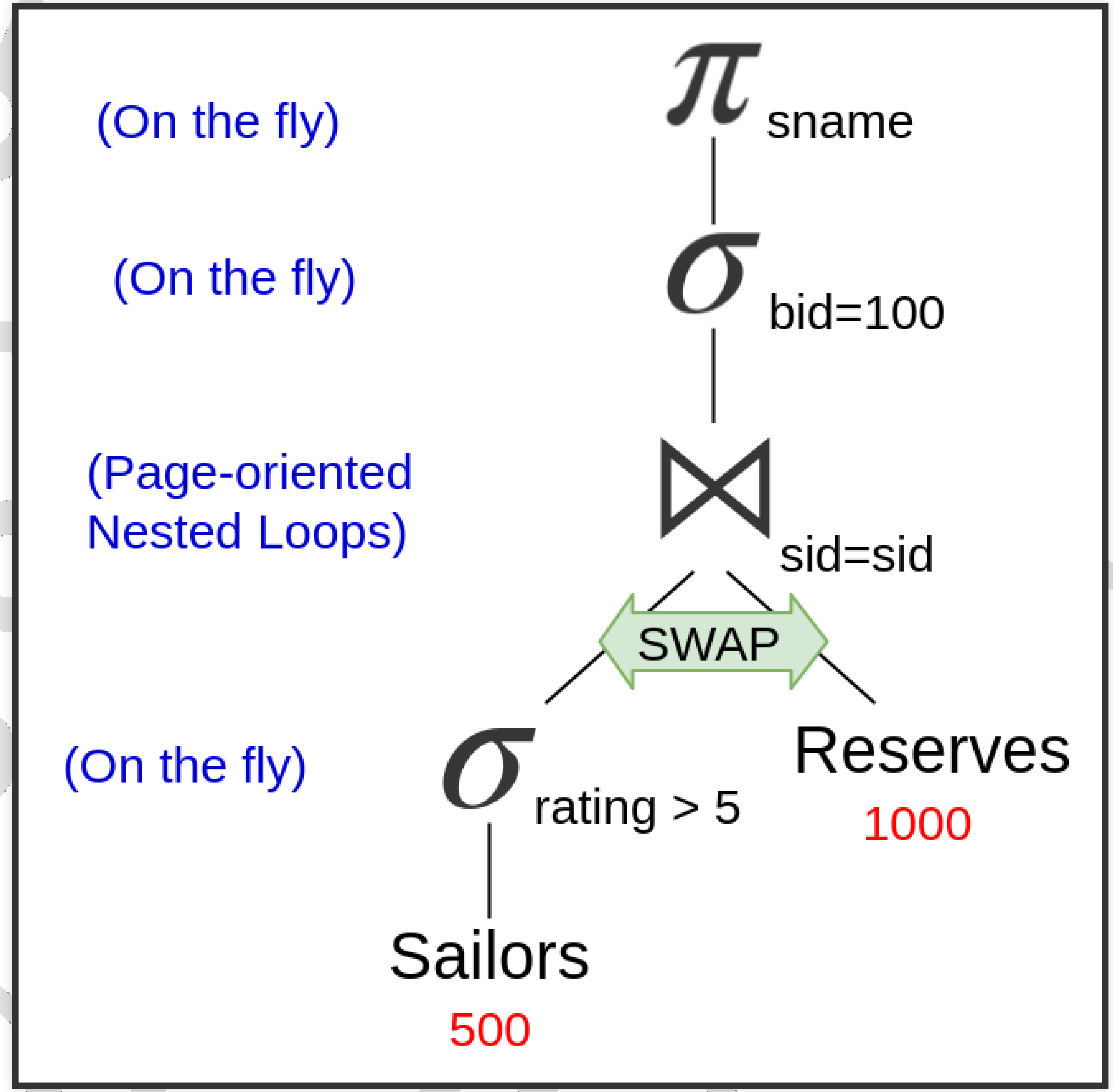
MOTIVATING EXAMPLE



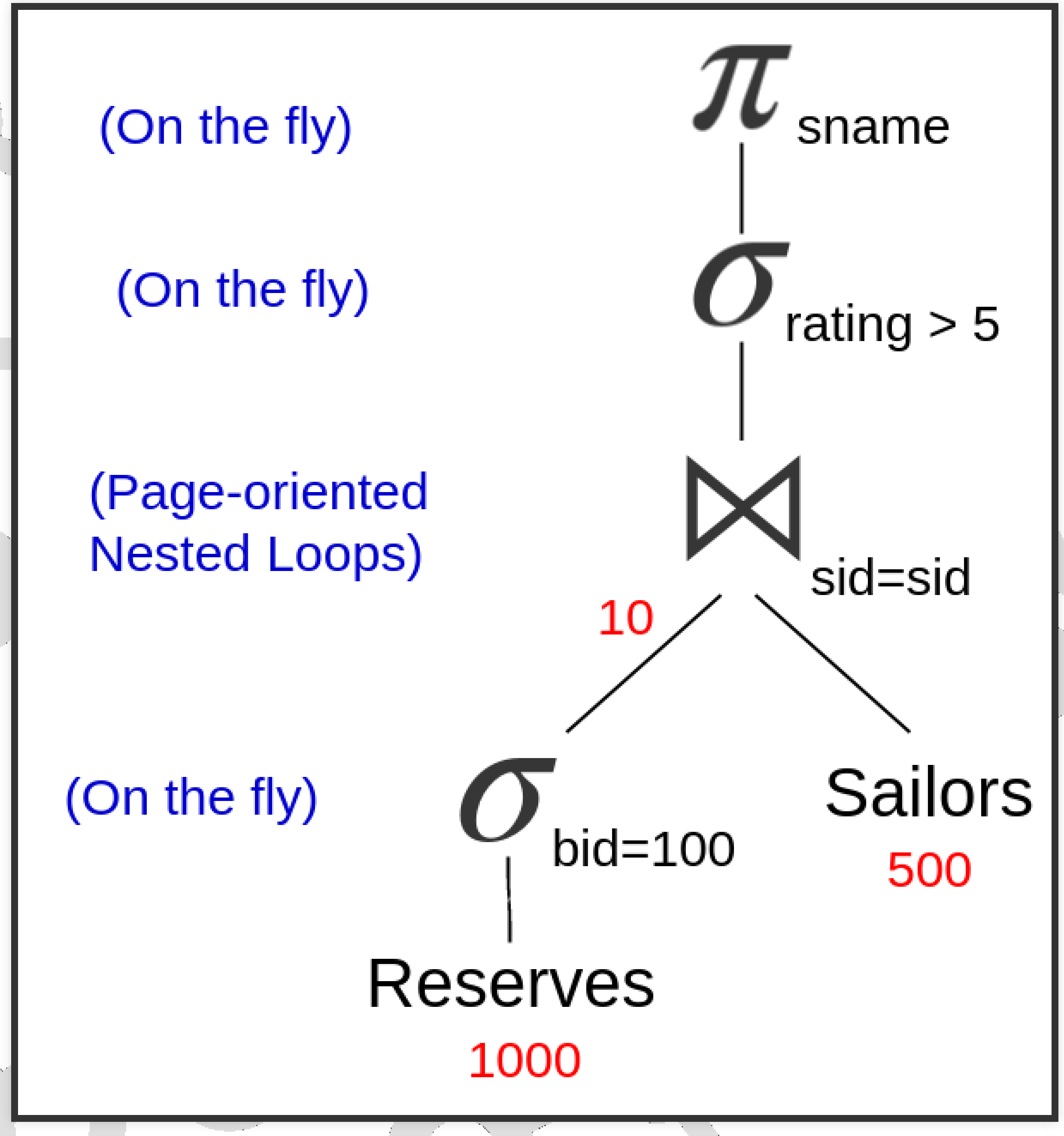
MOTIVATING EXAMPLE



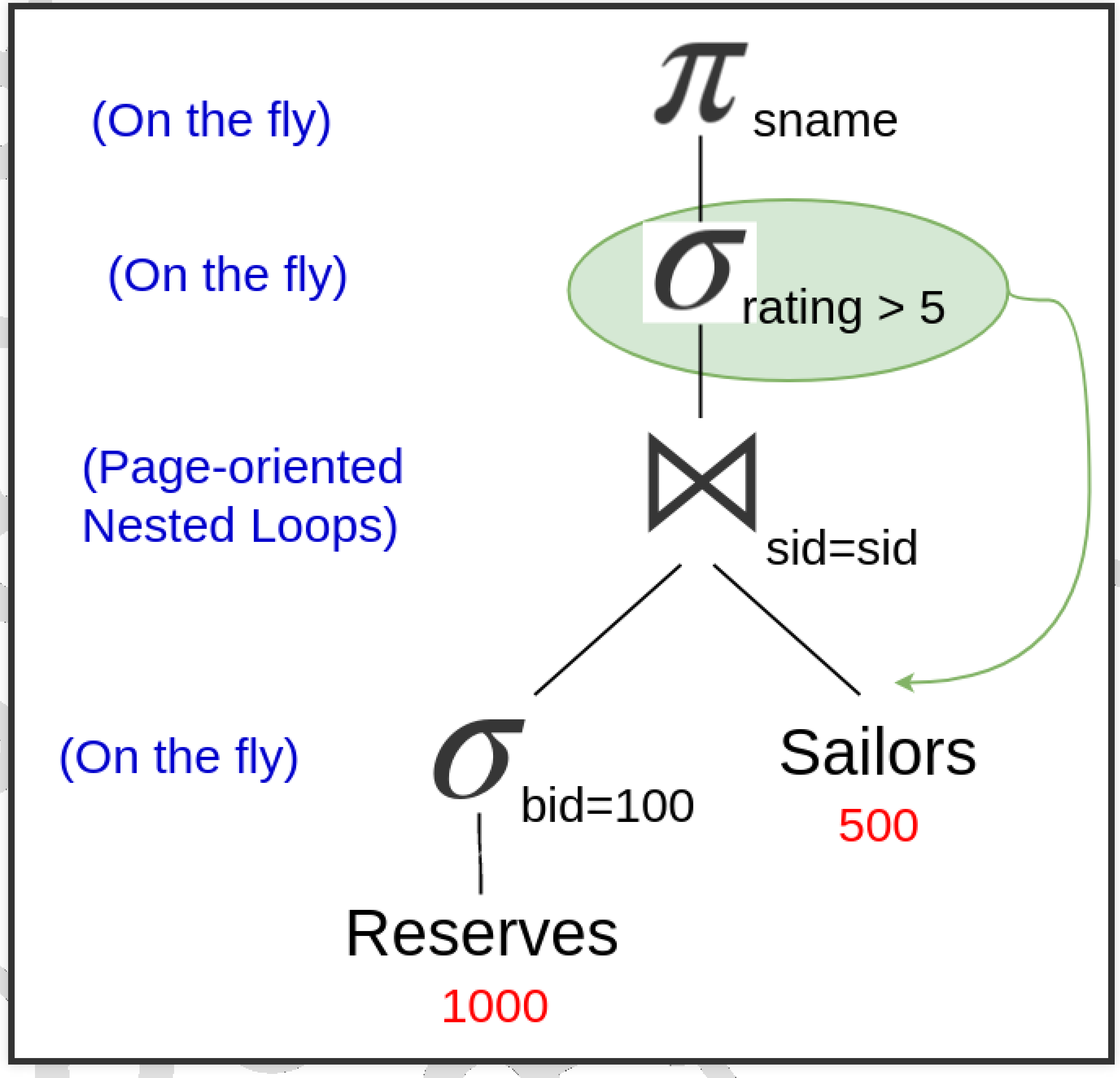
MOTIVATING EXAMPLE



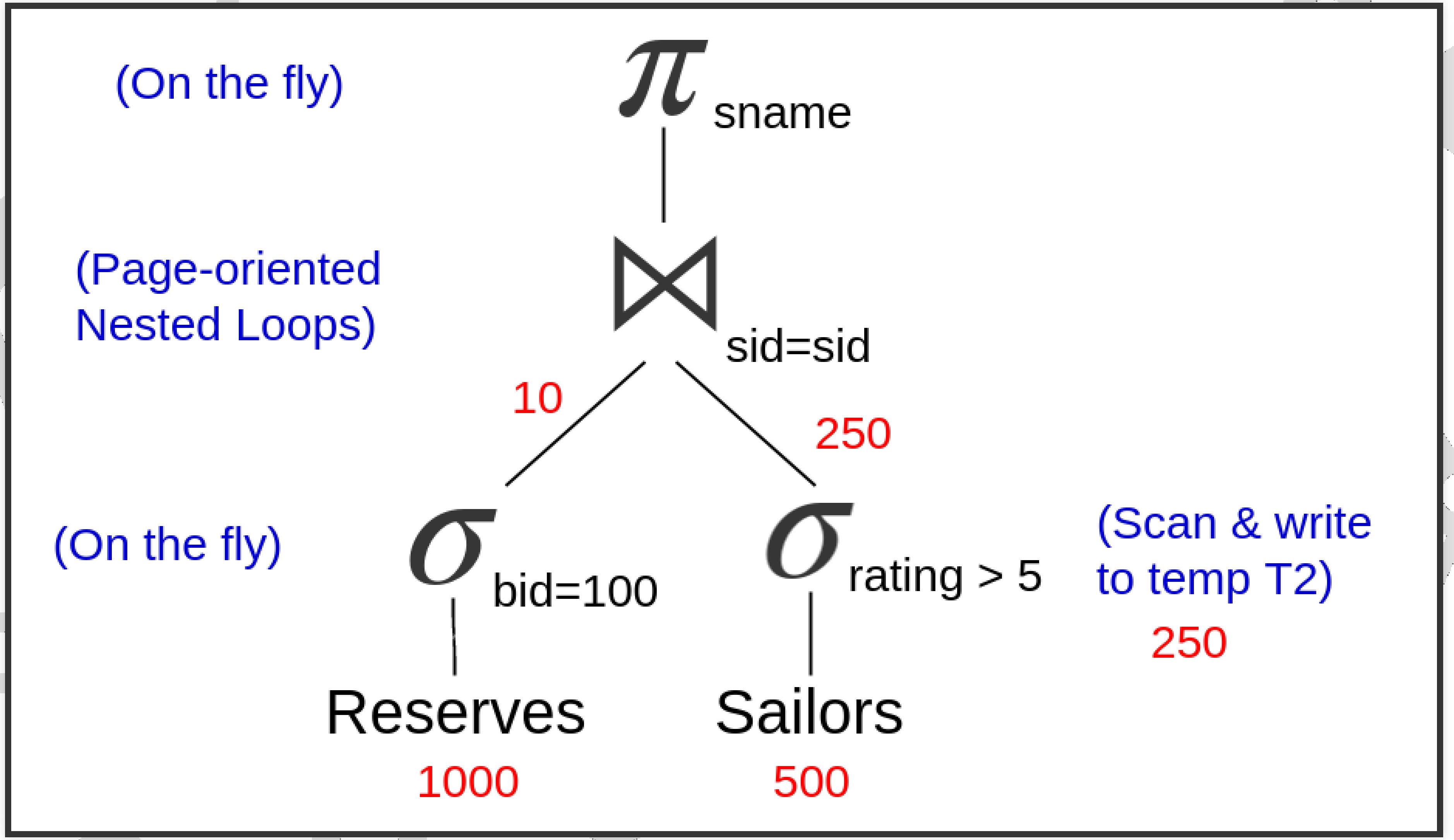
MOTIVATING EXAMPLE



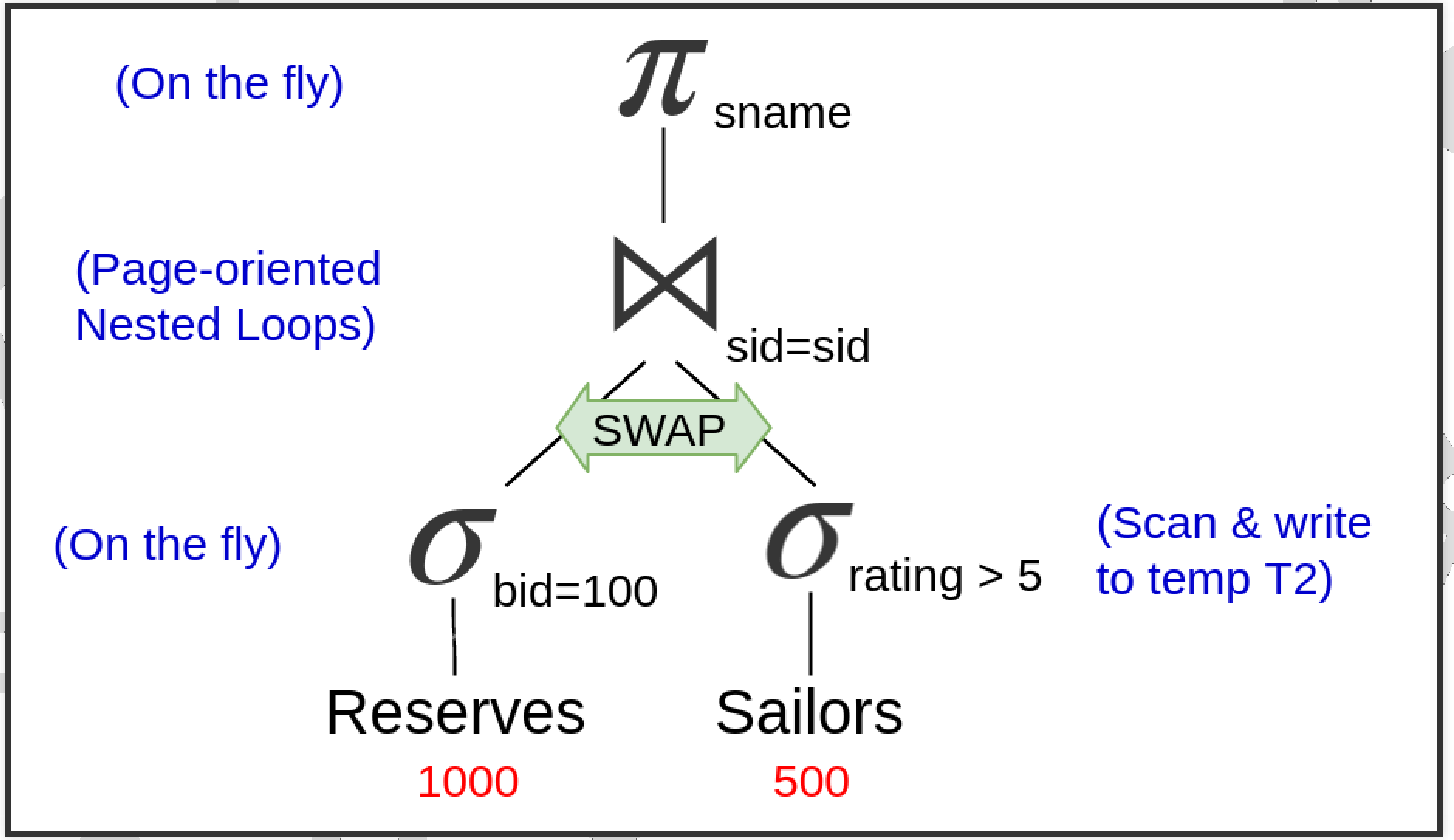
MOTIVATING EXAMPLE



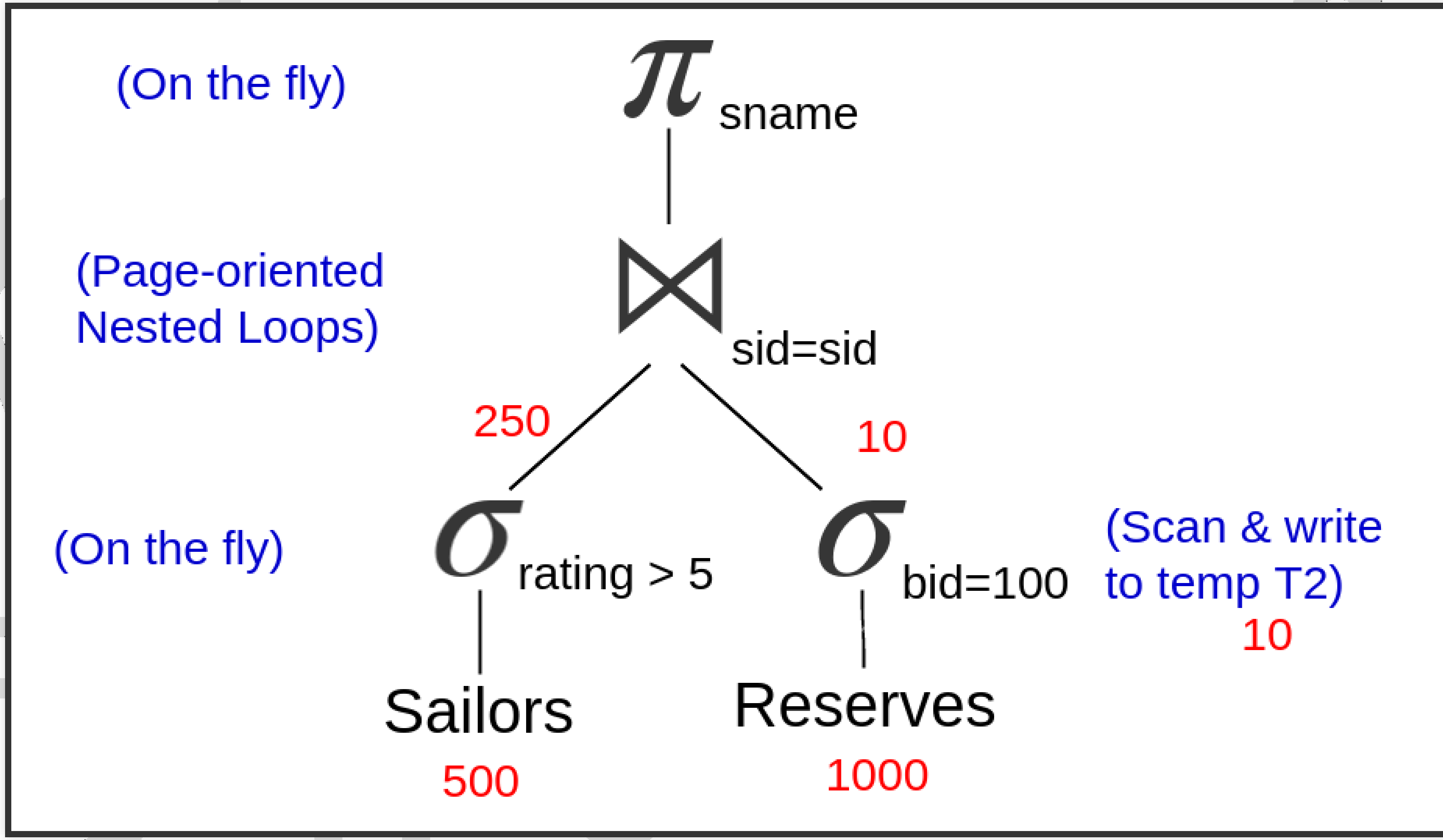
MOTIVATING EXAMPLE



MOTIVATING EXAMPLE

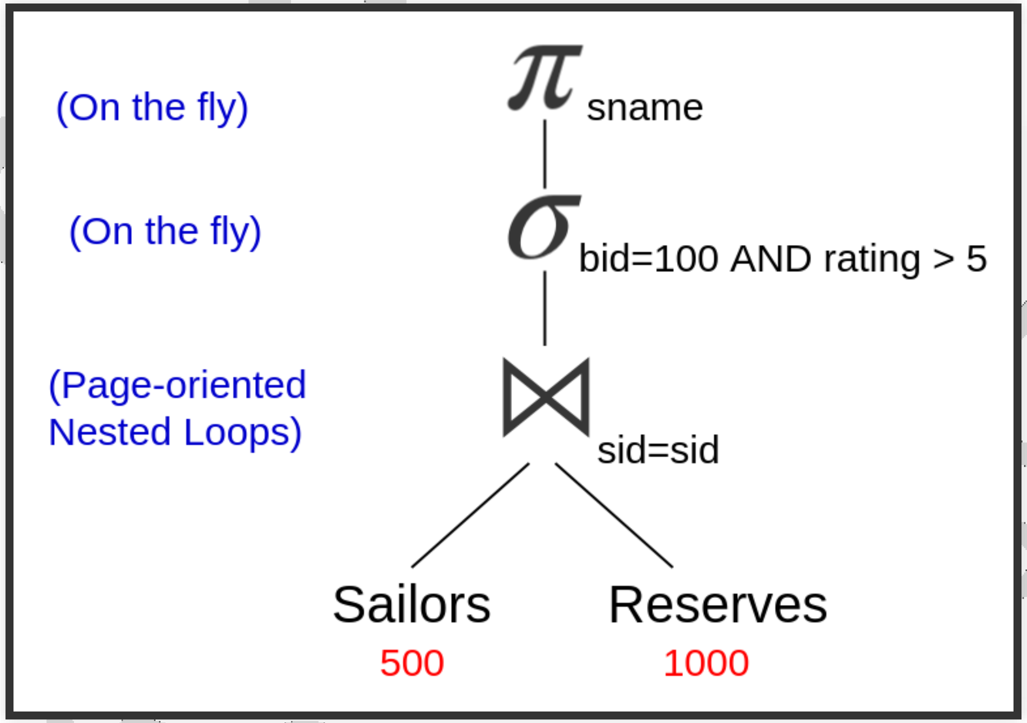


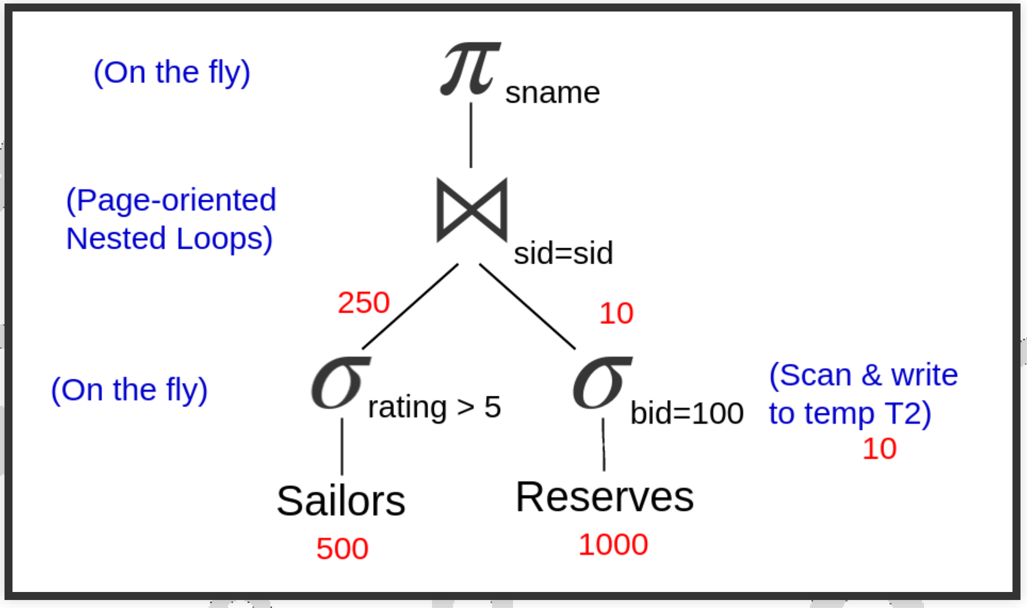
MOTIVATING EXAMPLE



MOTIVATING EXAMPLE

Original vs. optimized





Optimized query is **124x cheaper** than the original!

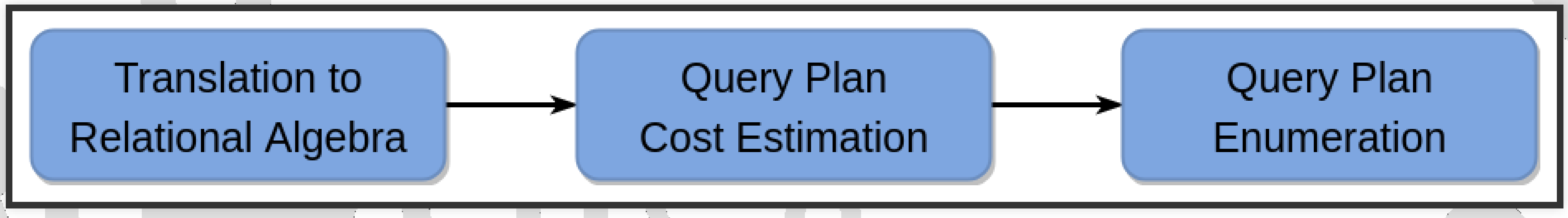
REVIEW SO FAR

- We know that there are different ways to implement an operation
  - physical operators
- We have studied how to estimate the cost of operators individually
- We also know operators could be executed in different order
  - relational algebra equivalence
- Choosing a query plan:
  - access path (index, file-scan)
  - physical operator
  - operator ordering

PROBLEMS TO BE SOLVED FOR IMPLEMENTING A QUERY OPTIMIZER

- How to estimate the cost of a query plan?
  - Size estimation and reduction factors
- Cost estimation needs statistics.
  - What statistics?
  - How to maintain them?
- How to generate alternate plans and what type of alternative plans?
- How to choose the “best” plan?

ROADMAP


Translation to Relational Algebra
Query Plan Cost Estimation
Query Plan Enumeration

HIGHLIGHTS OF SYSTEM R OPTIMIZER

- **Impact:**
  - Most widely used currently; works well for < 10 joins.
- **Cost estimation:** very inexact, but works in practice
  - Considers combination of CPU and I/O costs.
  - More sophisticated techniques known now.
- **Statistics:** maintained in system catalogs
- **Alternative plans:** only the space of *left-deep plans* is considered.
  - Cartesian products avoided in some implementations.
- **The "best" plan:** searched by *dynamic programming*

COST ESTIMATION

> To estimate the cost of query plan, we have to first estimate the cost of each operator

- The cost of an operator depends on the size of the input
- the output of one operator is the input of its parent

How to estimate the output size of an operator before we actually run it?

- In general, it is a guess of the Reduction Factor (RF)
  - **RF = output_size/input_size**
- Accuracy depends on what information we know about the input relations

THE GUESSING GAME OF RF

For an equality selection (column = value), if we know...

- only # tuples and # pages of the input:
  - System R's guess: 1/10
- If there is an index on column
  - and we know `NKeys`: # distinct key values in the index
  - System R's guess: `1/NKeys`

THE GUESSING GAME OF RF

Now how about inequality selection (e.g. column > value).

- If we only know # tuples and # pages of the input:
  - System R's guess: *a random fraction smaller than 1/2*
- If there is an index on column
  - and we know *High/Low*: the highest/lowest key values
  - System R's guess: (High – value)/(High-Low)

Q: What is the assumption of the above guesses with index?

POSTGRESQL DEFAULTS (1)

src/include/utils/selfuncs.h

```
/*
 * Note: the default selectivity estimates are not chosen entirely at random.
 * We want them to be small enough to ensure that indexscans will be used if
 * available, for typical table densities of ~100 tuples/page.  Thus, for
 * example, 0.01 is not quite small enough, since that makes it appear that
 * nearly all pages will be hit anyway.  Also, since we sometimes estimate
 * eqsel as 1/num_distinct, we probably want DEFAULT_NUM_DISTINCT to equal
 * 1/DEFAULT_EQ_SEL.
 */

/* default selectivity estimate for equalities such as "A = b" */
#define DEFAULT_EQ_SEL        0.005

/* default selectivity estimate for inequalities such as "A < b" */
#define DEFAULT_INEQ_SEL  0.3333333333333333

/* default selectivity estimate for range inequalities "A > b AND A < c" */
#define DEFAULT_RANGE_INEQ_SEL        0.005
```

POSTGRESQL DEFAULTS (1)

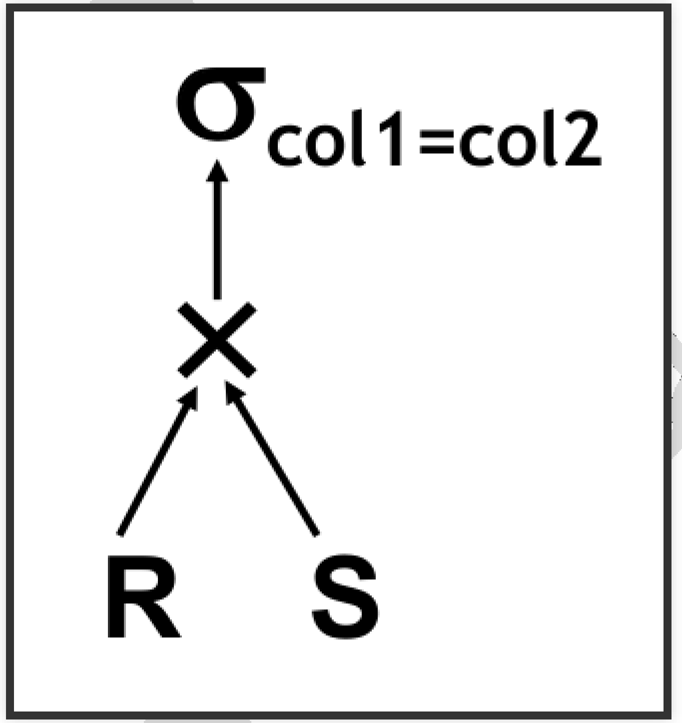
src/include/utils/selfuncs.h

```
/* default selectivity estimate for pattern-match operators such as LIKE */
#define DEFAULT_MATCH_SEL        0.005

/* default number of distinct values in a table */
#define DEFAULT_NUM_DISTINCT  200

/* default selectivity estimate for boolean and null test nodes */
#define DEFAULT_UNK_SEL                        0.005
#define DEFAULT_NOT_UNK_SEL                (1.0 - DEFAULT_UNK_SEL)
```

REDUCTION FACTORS FOR JOINS

RF for the term `col1=col2`

Its input size is the size of the cartesian product: $|R| \times |S|$



REDUCTION FACTORS FOR JOINS

Estimation of RF for Joins (term col1=col2)

If we have indices on both columns

- estimate each tuple r of R generates `NTuples(S)/Nkeys(S)` result tuples, so `|R|*|S|/Nkeys(S)`
- starting from S, yielding: `|S|*|R|/Nkeys( R )`
- If these two estimates differ, take the lower one!
  - Q: Why?

So RF = 1/MAX(NKeys(I1), NKeys(I2))

REDUCTION FACTORS FOR JOINS

The above estimations assume data are uniformly distributed. What if it is not true?

Use histogram to approximate the data distribution: Captures #tuples in ranges of values

REDUCTION FACTORS FOR JOINS

Table 1. equiwidth

No. of values	2	3	3	1	8	2	1
Value	0-0.99	1-1.99	2-2.99	3-3.99	4-4.99	5-5.99	6-6.99

Table 2. equidepth

No. of values	2	3	3	3	3	2	4
Value	0-0.99	1-1.99	2-2.99	3-4.05	4.06-4.67	4.68-4.99	5-6.99

COST ESTIMATION

For each plan considered, must estimate total cost:

- Must estimate cost of each operation in plan tree.
  - Depends on input cardinalities.
- Must estimate size of result for each operation in tree!
  - Use information about the input relations.
  - For selections and joins, assume independence of predicates.
- In System R, cost is boiled down to a single number consisting of #I/O + factor * #CPU instructions

Q: Is “cost” the same as estimated “run time”?

STATISTICS AND CATALOGS

Store necessary information about the relations and indexes in the catalogs:

- # tuples (NTuples) and # pages (NPages) per rel'n.
- # distinct key values (NKeys) for each index.
- low/high key values (Low/High) for each index
- Index height (IHeight) for each tree index.
- # index pages (INPages) for each index.

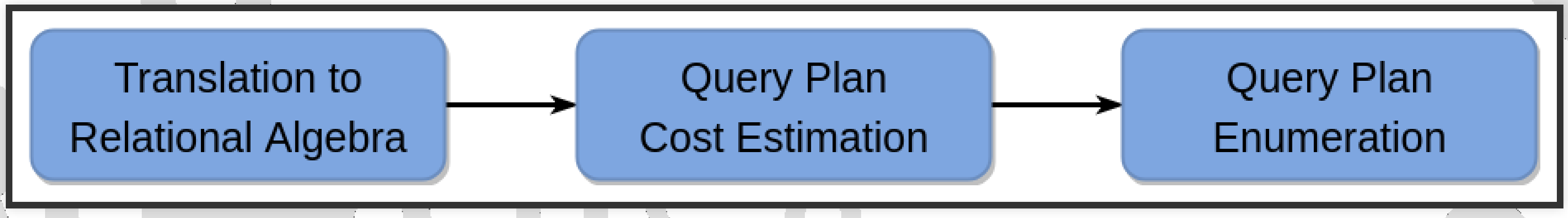
STATISTICS AND CATALOGS

Catalogs updated periodically.

- Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.

More detailed information (e.g., histograms of the values in some field) are sometimes stored.

ROADMAP


Translation to Relational Algebra
Query Plan Cost Estimation
Query Plan Enumeration

ENUMERATION OF ALTERNATIVE PLANS

There are two main cases:

- Single-relation plans
- Multiple-relation plans

ENUMERATION OF ALTERNATIVE PLANS

For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:

- Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
- The different operations are essentially carried out together
  - e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are pipelined into the aggregate computation.

COST ESTIMATES FOR SINGLE-RELATION PLANS

- Index I on primary key matches selection:
  - Cost is Height(I) + 1 for a B + tree.
- Clustered index I matching one or more selects:
  - (NPages(I)+NPages(R)) * product of RF's of matching selects.
- Non-clustered index I matching one or more selects:
  - (NPages(I)+NTuples(R)) * product of RF's of matching selects.
- Sequential scan of file:
  - NPages(R).

Must also charge for duplicate elimination if required

EXAMPLE

```sql
SELECT S.sid
FROM Sailors S
WHERE S.rating=8
```

If we have an index on rating:

- Cardinality = (1/NKeys(I)) * NTuples(R) = (1/10) * 40000 tuples
- Clustered index: (1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500) = 55 pages are retrieved. (This is the cost.)
- Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000) = 401 pages are retrieved.

EXAMPLE

```sql
SELECT S.sid
FROM Sailors S
WHERE S.rating=8
```

If we have an index on sid:

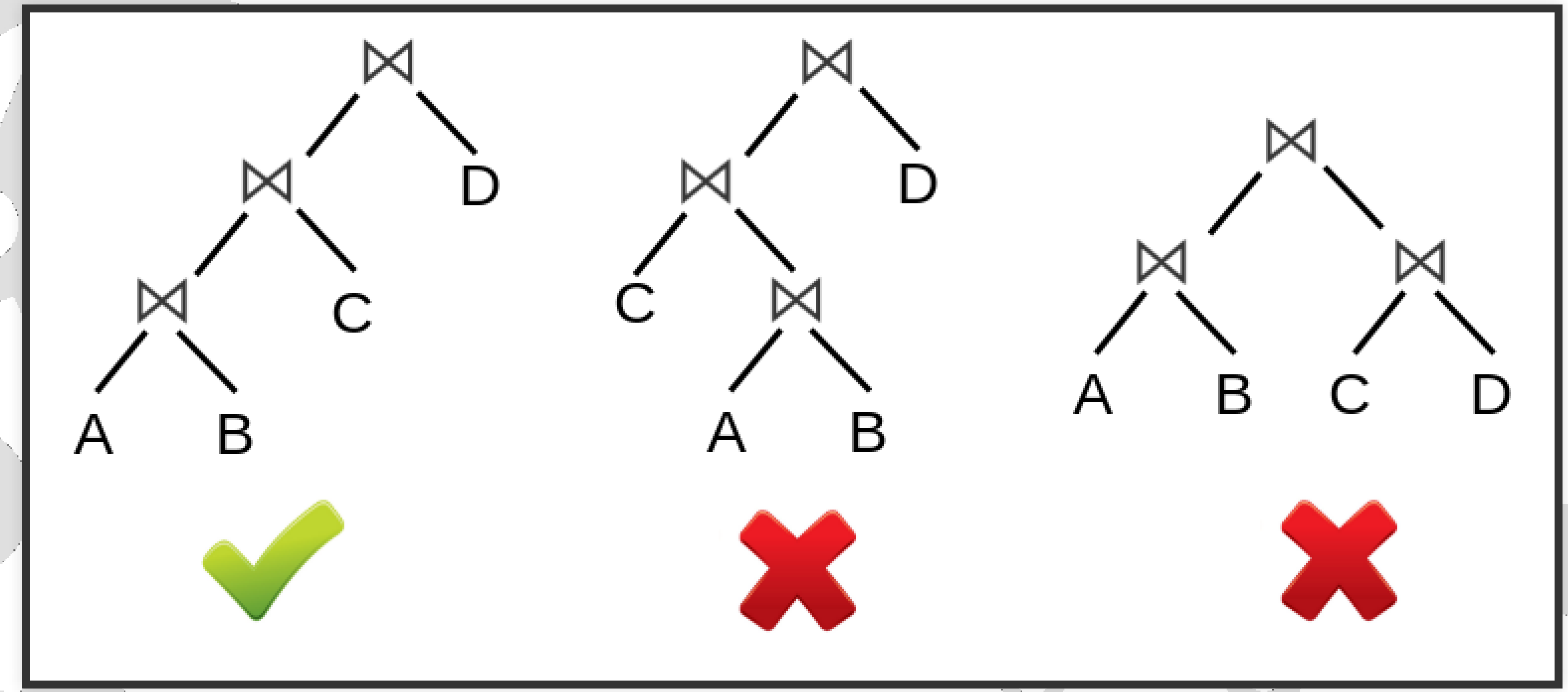
- Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000.

Doing a file scan:

- We retrieve all file pages (500).

QUERIES OVER MULTIPLE RELATIONS

A heuristic decision in System R: only left-deep join trees are considered.



QUERIES OVER MULTIPLE RELATIONS

As the number of joins increases, the number of alternative plans grows rapidly; we need to restrict the search space.

Left-deep trees allow us to generate all fully pipelined plans.

- Intermediate results not written to temporary files.
- Not all left-deep trees are fully pipelined (e.g., SM join).

ENUMERATION OF PLANS

Avoid Cartesian products:

- An N-1 way plan is not combined with an additional relation unless
  - there is a join condition between them,
  - or all predicates in WHERE have been used up.
- ORDER BY, GROUP BY, aggregates etc. handled as a final step,
  - using either an `interestingly ordered' plan
  - or an additional sort/hash operator.

In spite of pruning plan space, this approach is still exponential in the # of tables.

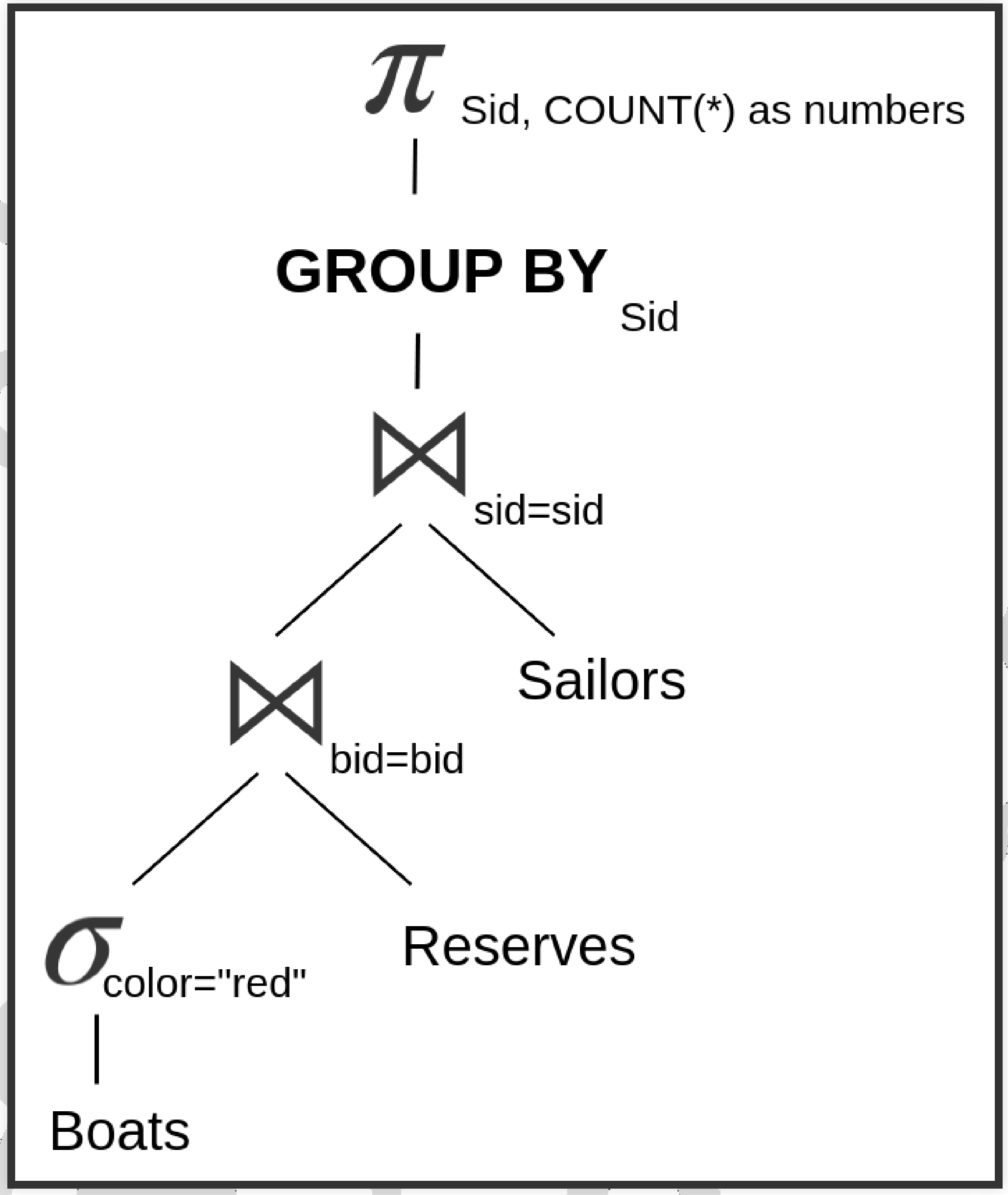
EXAMPLE

- Sailors: Hash, B+ on sid
- Reserves: Clustered B+ tree on bid, B+ on sid
- Boats: B+ on color, Hash on color

```sql
Select S.sid, COUNT(*) AS number
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid
AND B.color = “red”
GROUP BY S.sid
```

We should avoid generating the same ”sub-plans” repeatedly

EXAMPLE



π Sid, COUNT(*) as numbers
GROUP BY Sid
⋈ sid=sid
⋈ bid=bid
Sailors
σ color="red"
Reserves
Boats

PASS 1

- Sailors: Hash, B+ on sid
- Reserves: Clustered B+ tree on bid, B+ on sid
- Boats: B+ on color, Hash on color

Best plan for accessing each relation regarded as the first relation in an execution plan

- Reserves, Sailors: File Scan
- Boats: Hash on color

PASS 2

Take each of the plans in pass 1 as the outer, generate plans joining another relation as the inner, using all join methods (and matching inner access methods)

Retain cheapest plan for each pair of relations

Avoid cross-product.

PASS 2

- File Scan Reserves (outer) with Boats (inner)
- File Scan Reserves (outer) with Sailors (inner)
- File Scan Sailors (outer) with Boats (inner)
- File Scan Sailors (outer) with Reserves (inner)
- Boats hash on color with Sailors (inner)
- Boats hash on color with Reserves (inner)
  - e.g. Boats hash on color with Reserves B+tree on bid using Index Nested Loop Join
  - Boats hash on color with Reserves B+tree on bid using Sort Merge Join

PASS 3 AND BEYOND

For each of the plans retained from Pass 2, taken as the outer, generate plans for the next join

- eg Boats hash on color with Reserves (bid) (inner) (sort-merge)) inner Sailors (B-tree sid) sort-merge

Then, add the cost for doing the group by and aggregate:

- This is the cost to sort the result by sid, unless it has already been sorted by a previous operator.

Then, **choose the cheapest plan**

ENUMERATION OF LEFT-DEEP PLANS

Dynamic programming

Left-deep plans differ only in

- the order of relations,
- the access method for each relation,
- and the join method for each join.

ENUMERATION OF LEFT-DEEP PLANS

Dynamic programming

Enumerated using N passes (if N relations joined):

- Pass 1: Find best 1-relation plan for each relation.
- Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. (All 2-relation plans.)
- Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. (All N-relation plans.)

For each subset of relations, retain only:

- Cheapest plan overall, plus
- Cheapest plan for each interesting order of the tuples.

THE DYNAMIC PROGRAMMING TABLE

Subset of tables in FROM clause	Interesting-order columns	Best plan	Cost
{R,S}	<none>	hashjoin(R,S)	1000
{R,S}	<R.a,S.b>	sortmerge(R,S)	1500

INTERESTING ORDERS

An intermediate result has an "interesting order" if it is sorted by any of:

- ORDER BY attributes
- GROUP BY attributes
- Join attributes of *yet-to-be-added* (downstream) joins

SUMMARY

- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).

SUMMARY

Two parts to optimizing a query:

- Consider a set of alternative plans.
  - Good to prune search space; e.g., left-deep plans only, avoid Cartesian products.
- Must estimate cost of each plan that is considered.
  - Output cardinality and cost for each plan node.
  - Key issues: Statistics, indexes, operator implementations.

SUMMARY

Single-relation queries:

- All access paths considered, cheapest is chosen.
- Issues: Selections that match index, whether index key has all needed fields and/or provides tuples in a desired order.

SUMMARY

Multiple-relation queries:

- All single-relation plans are first enumerated.
  - Selections/projections considered as early as possible.
- Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered.
- Next, for each 2-relation plan that is `retained', all ways of joining another relation (as inner) are considered, etc.
- At each level, for each subset of relations, only the best plan for each interesting order of tuples is `retained'.

SUMMARY

- Optimization is the reason for the lasting power of the relational system
- But it is primitive in some ways
- New areas: Smarter summary statistics (fancy histograms and “sketches”), auto-tuning statistics, adaptive runtime re-optimization

REFERENCES

- https://doxygen.postgresql.org/selfuncs_8h.html