

Processing Top N Queries

Donald Kossmann

Overview

- **Motivation**
 - SQL Extension
- **Query Processing Techniques**
 - Implementation of Stop Operators
 - Query Optimization
- **Performance Experiments and Results**
- **Related Work**
- **Summary**

Motivation

- Many applications require **top N** queries
- Example 1 - Web databases
 - *find the five cheapest hotels in Madison*
- Example 2 - Decision Support
 - *find the three best selling products*
 - *average salary of the 10,000 best paid employees*
 - *send the five worst batters to the minors*
- Example 3 - Multimedia / Text databases
 - *find 10 documents about „database“ and „web“.*
- Queries and updates, any N, all kinds of data

Key Observation

Top N queries cannot be expressed *well* in SQL

```
SELECT * FROM Hotels h
WHERE city = Madison AND
5 > (SELECT count(*) FROM Hotels h1
WHERE city = Madison AND h1.price < h.price);
```

- So what do you do?
 - Implement top N functionality in your application
 - Extend SQL and the database management system

Implementation of Top N in the App

- Applications use SQL to get as close as possible
- Get results *ordered*, consume only N objects and/or specify *predicate* to limit # of results

```
SELECT *
```

```
FROM Hotels
```

```
WHERE city = Madison
```

```
ORDER BY price;
```

```
SELECT *
```

```
FROM Hotels
```

```
WHERE city = Madison
```

```
AND price < 70;
```

- either too many results, **poor performance**
- or not enough results, **user must ask query again**
- difficult for nested top N queries and updates

Extend SQL and DBMS

- **STOP AFTER** clause specifies number of results

```
SELECT *  
FROM Hotels  
WHERE city = Madison  
ORDER BY price  
STOP AFTER 5 [WITH TIES];
```

- Returns five hotels (plus ties)
- Challenge: extend query processor, performance

Updates

- Give **top 5** salesperson a 50% salary raise

```
UPDATE Salesperson SET salary = 1.5 * salary
WHERE id IN
(SELECT id
FROM Salesperson
ORDER BY turnover DESC
STOP AFTER 5);
```

Nested Queries

- The average salary of the **top 10000** Emps

```
SELECT AVG(salary)
FROM (SELECT      salary
      FROM      Emp
      ORDER BY   salary DESC
      STOP AFTER 10000);
```

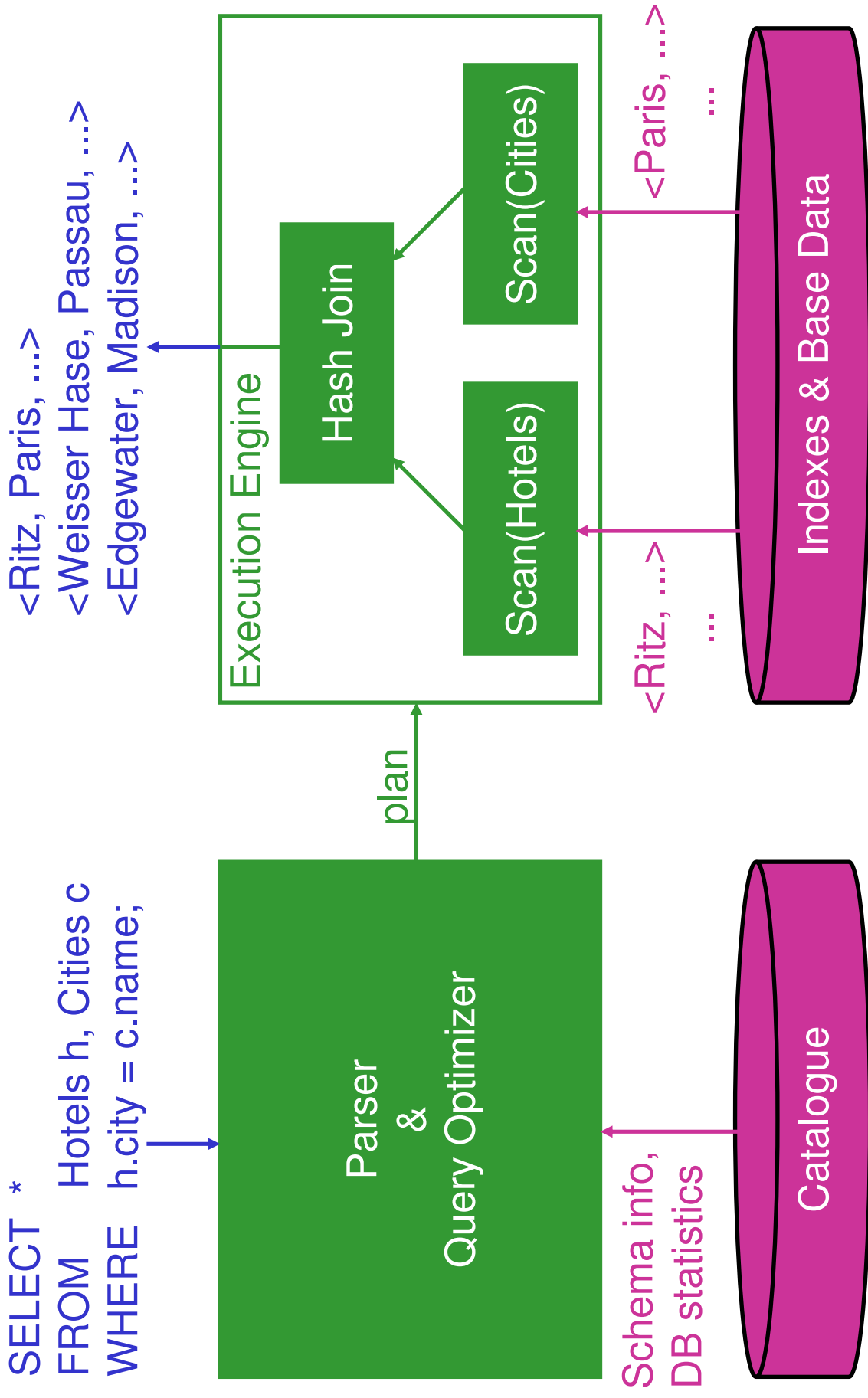
Extend SQL and DBMSs

- SQL syntax extension needed
- All major database vendors do it
- Unfortunately, everybody uses a different syntax
 - Microsoft: *set rowcount N*
 - IBM DB2: *fetch first N rows only*
 - Oracle: *rownum < N* predicate
 - SAP R/3: *first N*
- Challenge: extend query processor of a DBMS

Overview

- Motivation
 - SQL Extension
- Query Processing Techniques
 - Implementation of Stop Operators
 - Query Optimization
- Performance Experiments and Results
- Related Work
- Summary

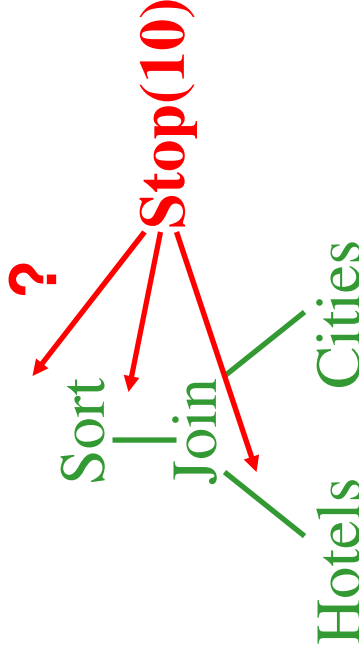
Query Processing 101



Processing Top N Queries

- Overall goal: avoid wasted work
- **Stop** operators encapsulate **top N** operation
 - implementation of other operators not changed
- **Extend optimizer** to produce plans with **Stop**

```
SELECT *  
FROM Hotels h, Cities c  
WHERE h.city = c.name  
ORDER BY h.price  
STOP AFTER 10;
```



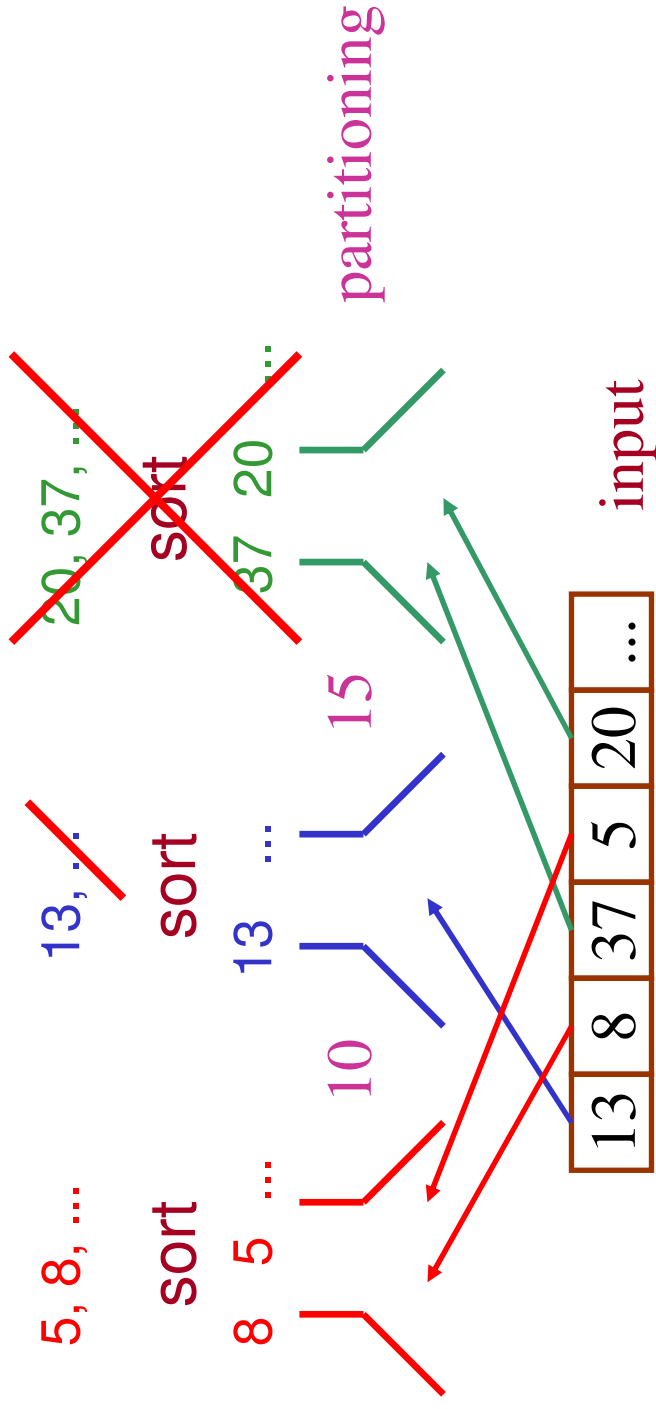
Implementation of Stop Operators

- There are several alternative ways to implement Stop operators
- Performance of these implementations depends on:
 - N
 - availability of indexes
 - size of available main memory
 - properties of other operations in the query

Implementation Variants

- **Stop** after a **Sort** (trivial)
- **Priority queue**
 - build main memory priority queue with first N objects of input
 - read other objects one at a time:
 - test membership bounds & replace**
- **Partition** the input (range-based braking)
- **Stop** after an **Index-Scan**

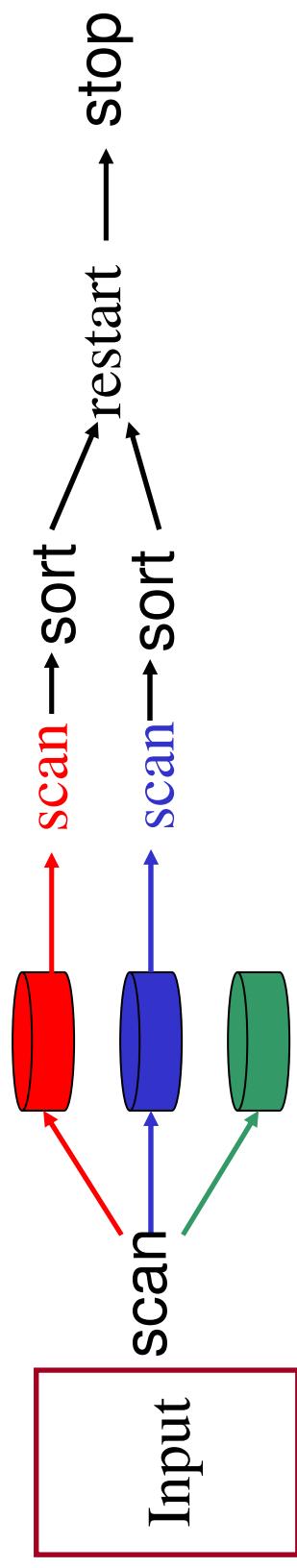
Range-based Braking



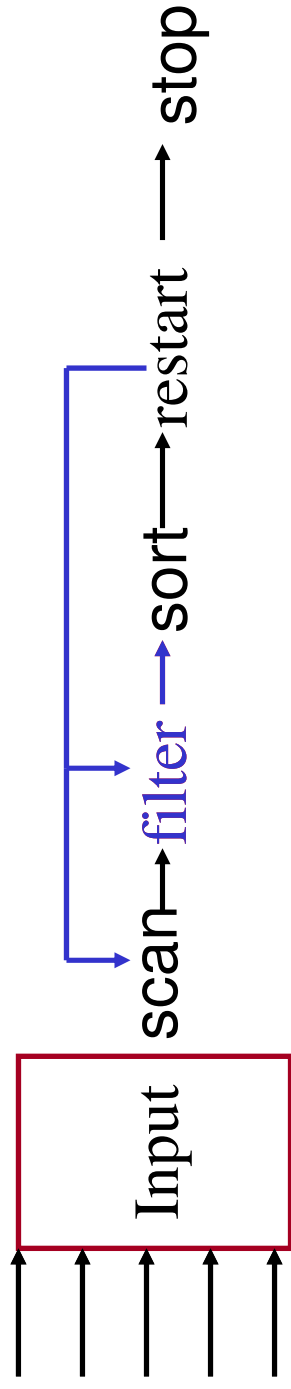
- Adapt ideas from parallel sorting [DeWitt&Naughton]
- Use histograms (if available) or sampling

Range-based Braking Variants

1. Materialize: store all partitions on disk



2. Reread: scan input for each partition



3. Hybrid: materialize first x partitions; reread others

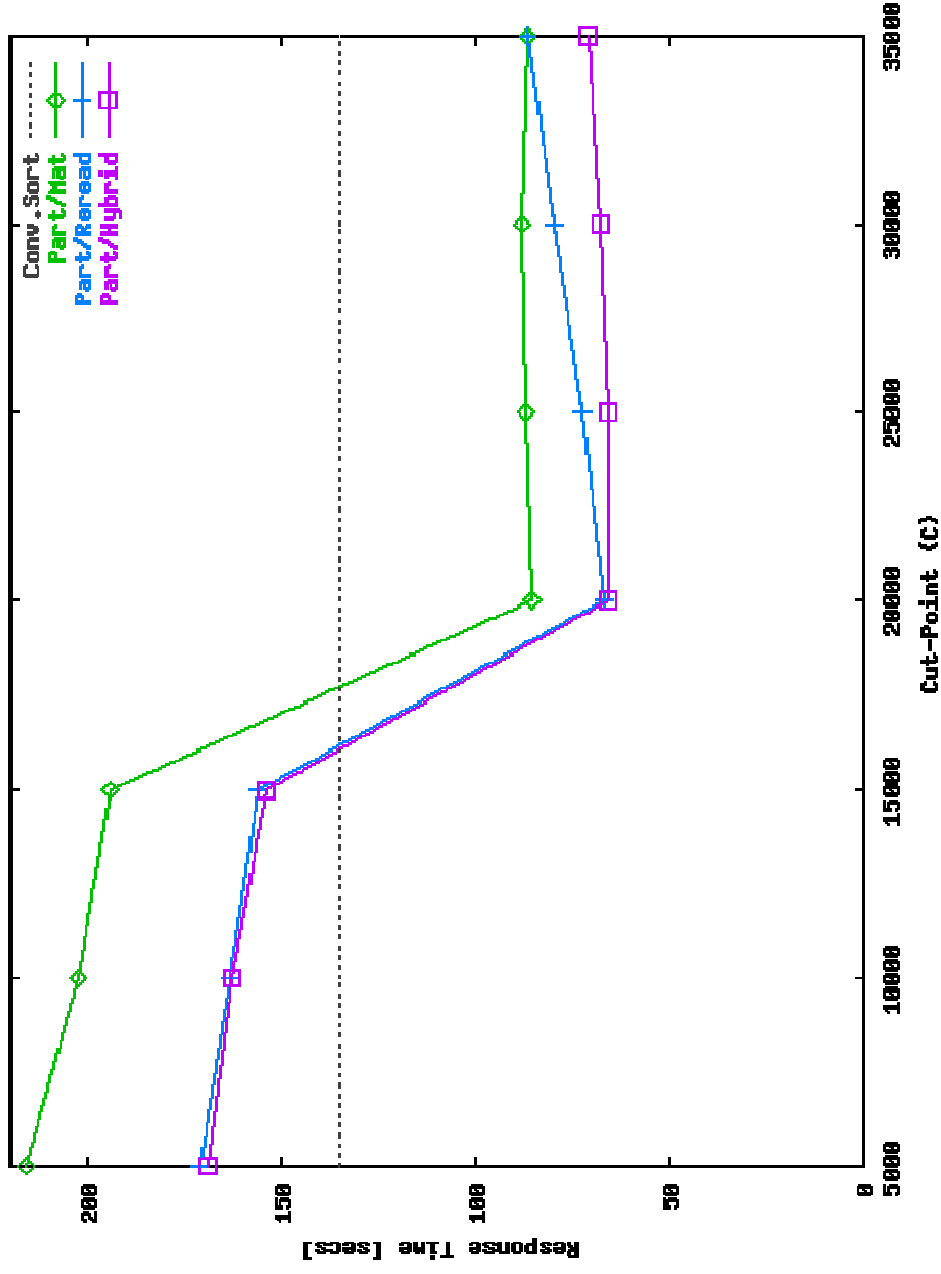
Performance of Stop Operators

N highest paid Emps; AODB/Sun; 4 MB mem.; 50 MB DB

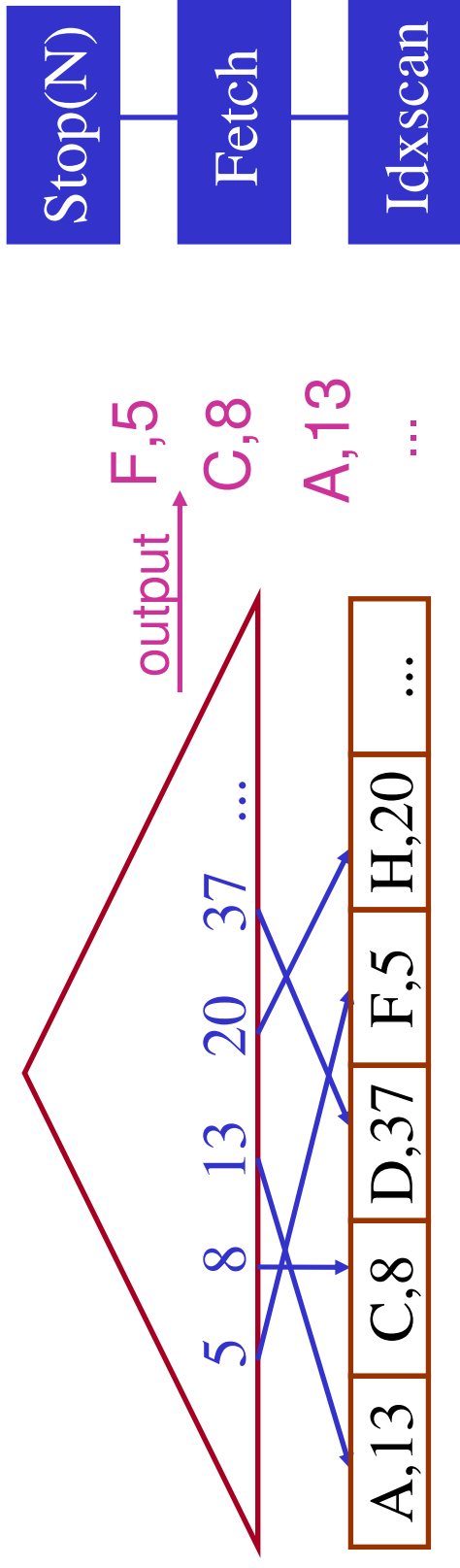
	N	100	50K	ALL
Sort	104.2	103.2	112.2	117.9
PQ	54.0	52.7	n.a.n	n.a.n
Mat	75.3	75.0	83.6	120.1
Reread	50.0	50.1	83.6	120.1
Hybrid	49.5	50.0	87.6	126.4

Range-based Braking

Sensitivity Analysis

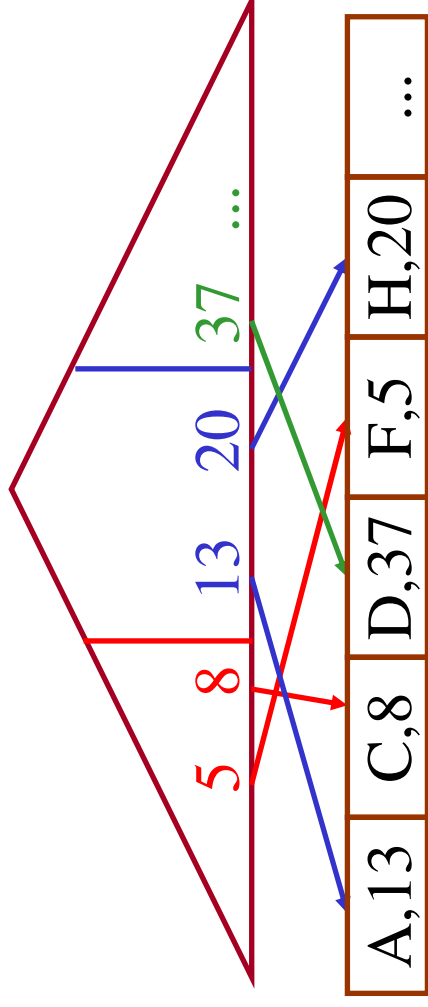


Stop & Indexes

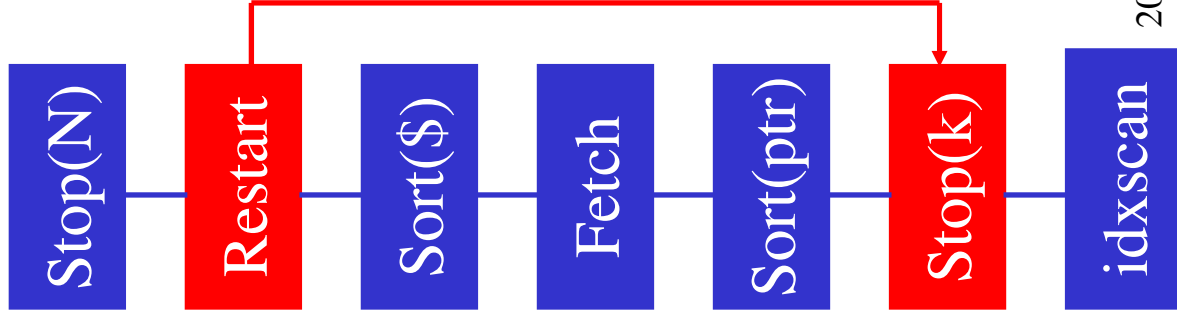


- Read and follow pointers from index until N results have been produced
- Very simple to implement, result is sorted
- Random I/O if N is large or if there is an additional predicate (e.g., hotels in Madison)

Range-based Braking & Indexes



- read **first** partition
- sort pointers to avoid random I/O
- read objects using (sorted) pointers
- re-sort tuples
- repeat until N results are produced



Performance Evaluation (Index)

N highest paid Emps; AODB/Sun; 4 MB mem.; 50 MB DB

N	10	1K	10K	50K
Index	1.8	92.8	807.4	4505.5
Part&Index	1.0	7.8	31.0	148.1
Hybrid	49.5	55.0	55.7	87.6

Summary: Stop Operators

- Small N ($N < 10K$)
 - use index with range-partitioning, if possible
 - use priority queues otherwise
- Large N ($N > 10K$)
 - use range-partitioning, hybrid variant
- Partitioning requires good histograms
 - additional cost of sampling affordable for large N
 - in some cases, get statistics on the fly for free
- Partitioning makes other plans possible

Overview

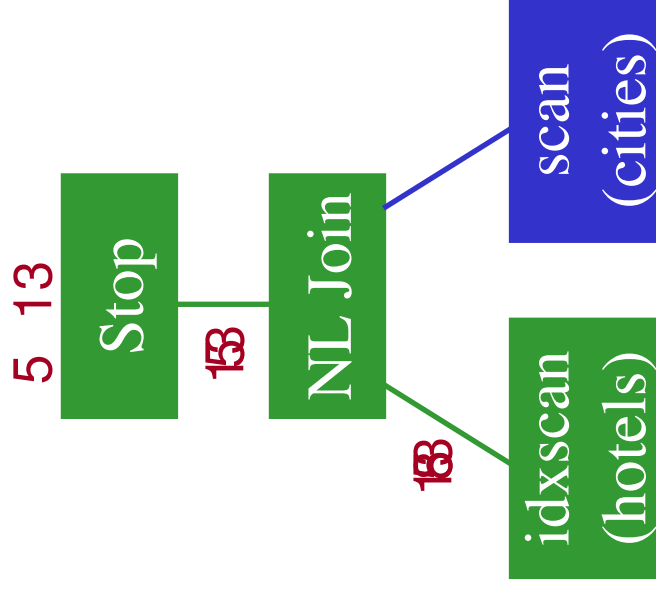
- Motivation
 - SQL Extension
- Query Processing Techniques
 - Stop Operators
 - **Query Optimization**
- Performance Experiments and Results
- Related Work
- Summary

Optimizing Top N Queries

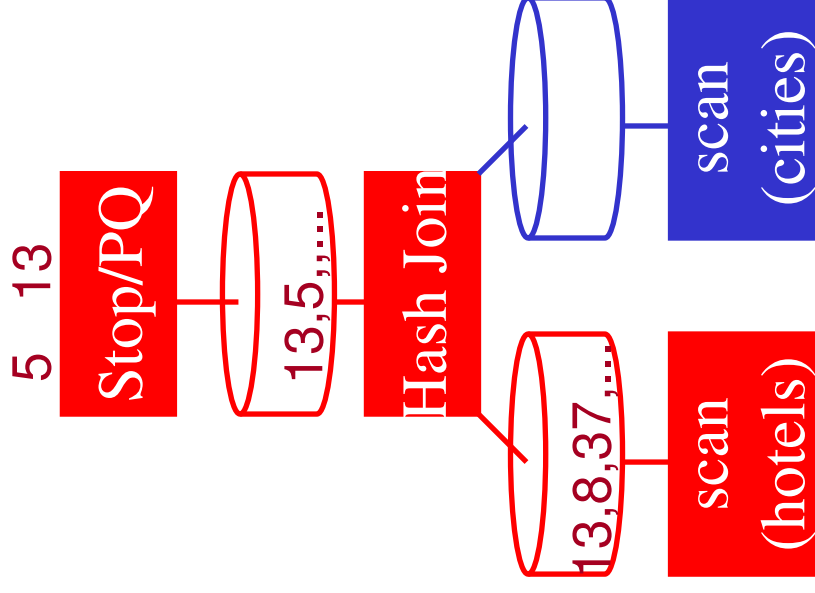
- Traditional optimizer must decide
 - join order
 - access paths (i.e., use of indexes), ...
- Top N optimizer must in addition decide
 - which implementation of Stop operator to use
 - where in a plan to place Stop operators
- Optimizer enumerates all alternative plans and selects best plan using a cost model
- Stop operators affect other decisions (e.g., join order, access paths)

Favor Pipelined Plans for Small N

- **pipelining operators** process a tuple at a time
- **blocking operators** consume whole input





5, 8, 13, 20, 37, ...

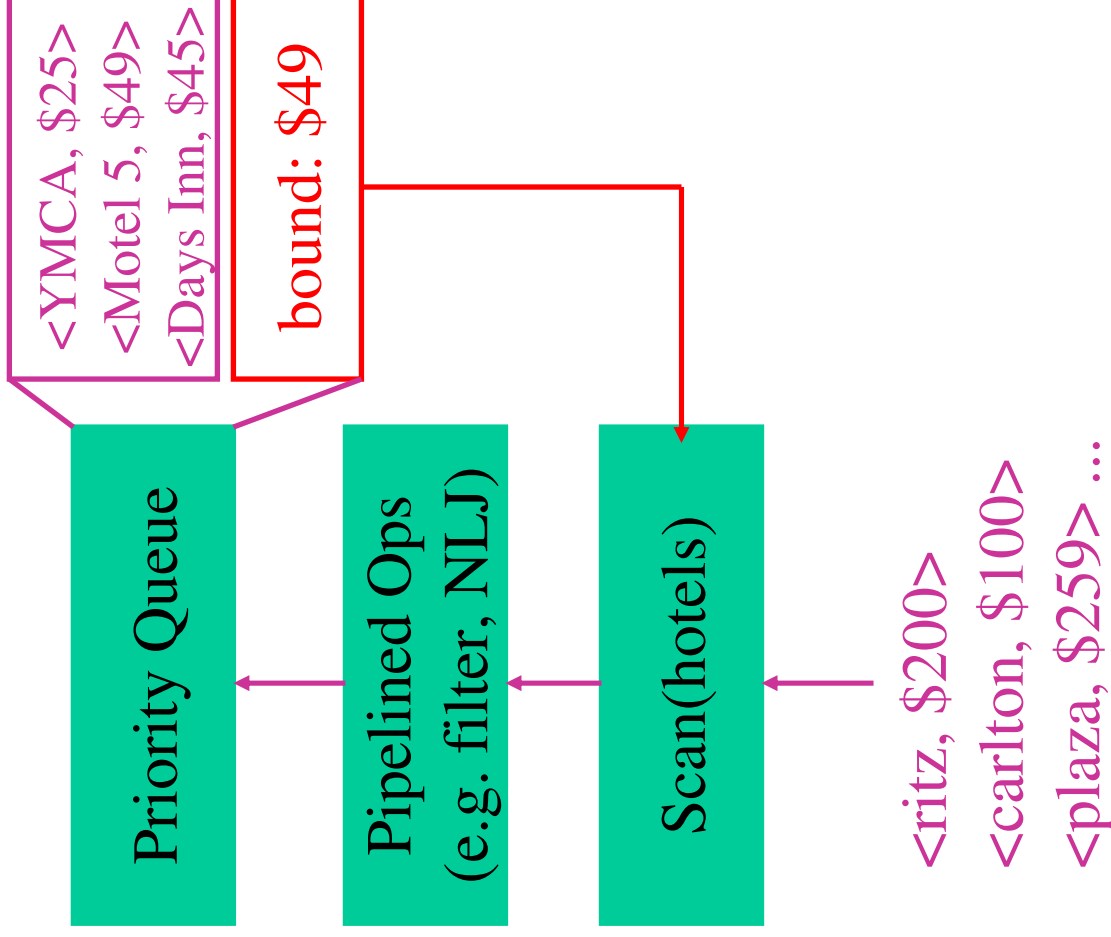


13, 8, 37, 5, 20, ...

Pipelined Plans: Costing

- Cost of a **pipelined plan** to produce N results:

$$CP(1) + N * p$$
- Cost of a **non-pipelined plan** to produce N results:

$$CM(\text{all}) + N * m$$
- typically: $CP(1) \ll CM(\text{all})$; $m \ll p$;
- **pipelined plan often wins if N is small**

Optimization of Stop in a Pipeline



Push Down Stop Operators

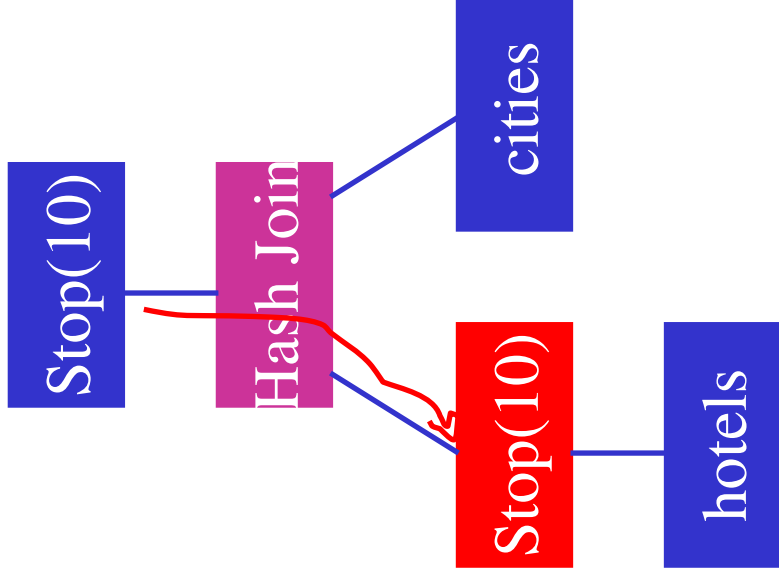
Through Pipeline Breakers

- Sometimes, **pipelined plan is not attractive**
- Or, **pipelined plan is not possible (no indexes)**
- In these cases, apply **Stop as early as possible** in order to reduce size of intermediate results
- Analogous to **predicate push-down** in traditional query optimization

Conservative Approach

- example:

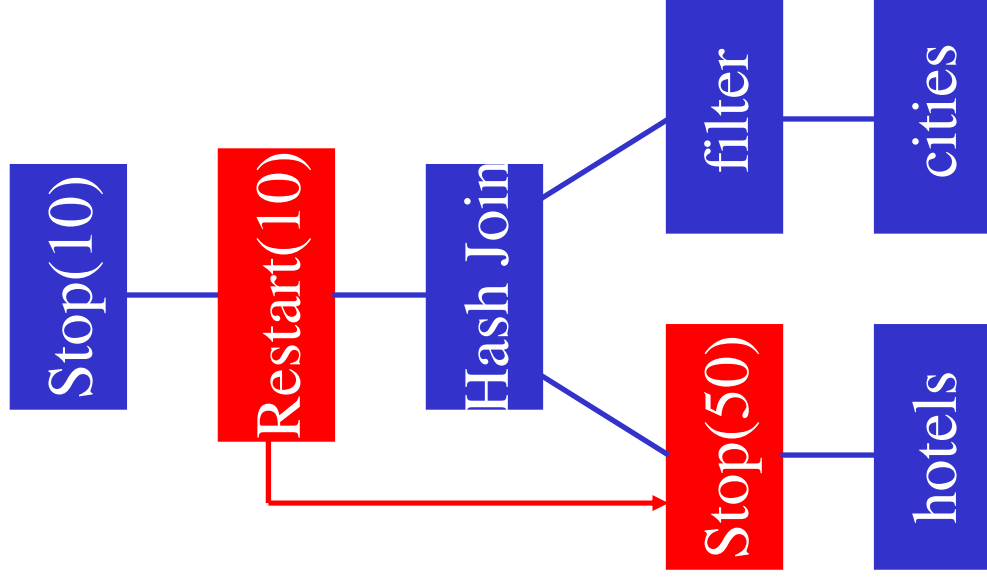
```
SELECT *  
FROM Hotels h, Cities c  
WHERE h.city = c.name  
ORDER BY price  
STOP AFTER 10;
```
- look at integrity constraints
- Push-down through **non-reductive** operators
- Every hotel qualifies join (**join is non-reductive**)
- **Stop** at the top necessary if a hotel matches several cities



Aggressive Approach

- Conservative approach not always applicable
- example:

```
SELECT *  
FROM Hotels h, Cities c  
WHERE h.city = c.name  
AND c.state = Wisconsin  
ORDER BY price  
STOP AFTER 10;
```
- partition on **price** before join
- use DB statistics

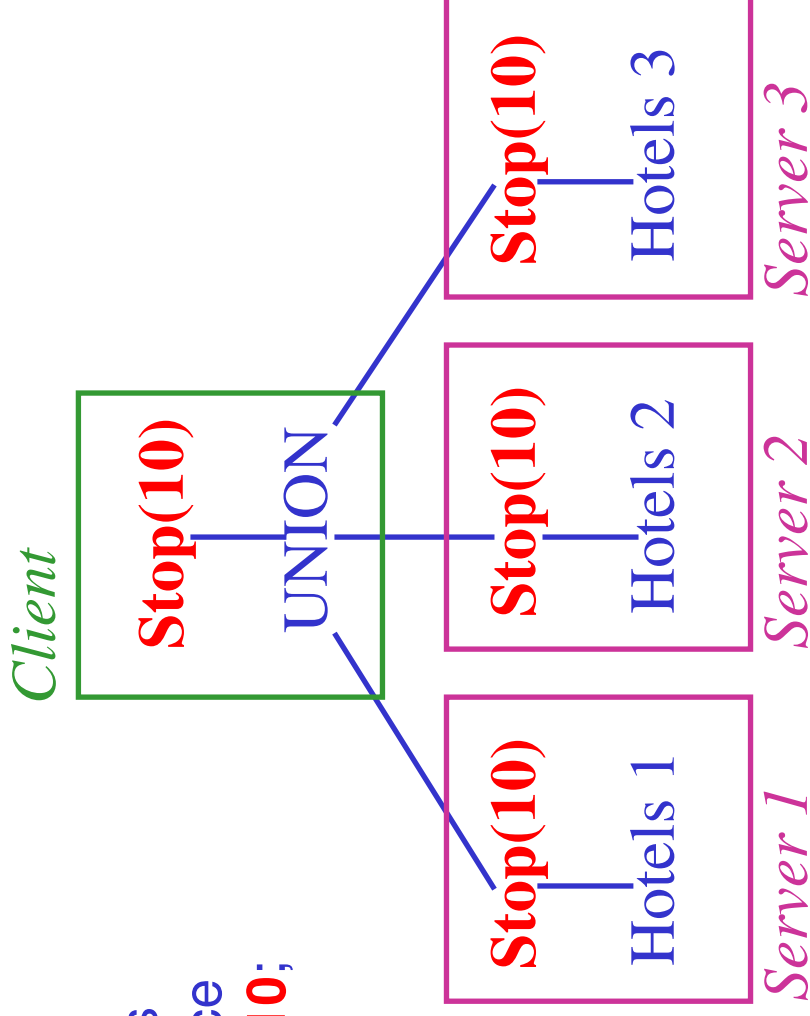


Conservative vs. Aggressive

- If Conservative applicable, do it.
- Aggressive:
 - can reduce the cost of other operations significantly (e.g., joins, sorts)
 - (unanticipated) restarts due to poor partitioning (i.e., bad statistics) cause additional costs
- Conservative is being implemented by IBM
- No commercial product is Aggressive yet

Union Queries (Parallel System)

```
SELECT *  
FROM Hotels  
ORDER BY price  
STOP AFTER 10;
```



Top N and Semi-joins

- Idea
 - keep rids, project out columns at the beginning
 - at the end use rids to refetch columns
- Tradeoff
 - reduces cost of joins, sorts etc. because intermediate results are smaller
 - additional overhead to refetch columns
- Attractive for top N because N limits refetch

Overview

- Motivation
 - SQL Extension
- Query Processing Techniques
 - Stop Operators
 - Query Optimization
- Performance Experiments and Results
- Related Work
- Summary

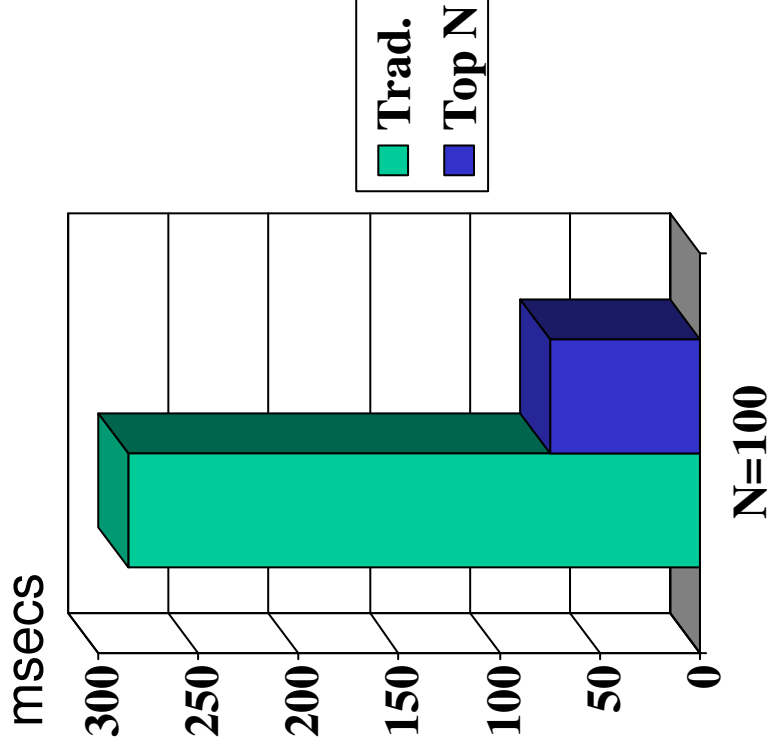
Performance Experiments

- Experimental Environment
 - (Simulation on) IBM DB2 V2.1 on RS 6000
 - AODB system on Sun SPARC station
- Benchmark
 - Emp, Dept, Travel database, vary size of database
 - Vary available indexes (none, clustered, unclustered)
 - Single-table, multi-way join queries, nested queries
 - Vary N
- Traditional system vs. Top N-enhanced system
- Conservative vs. Aggressive/range-partitioning

Experiment 1: Shipping Results

100 highest paid Emps, DB2, clustered index, 10 MB DB

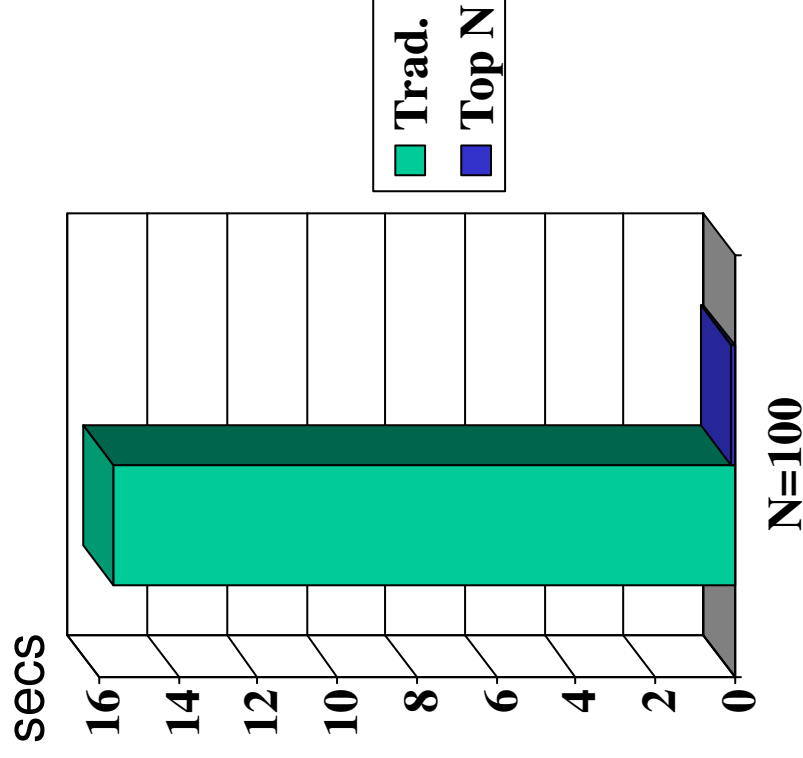
- Results are shipped in blocks of tuples in c/s environment
- traditional system produces and ships 512 tuples
- enhanced system produces and ships only N=100 tuples



Experiment 2: Access Path Selection

100 highest paid Emps, DB2, unclustered index, 10 MB DB

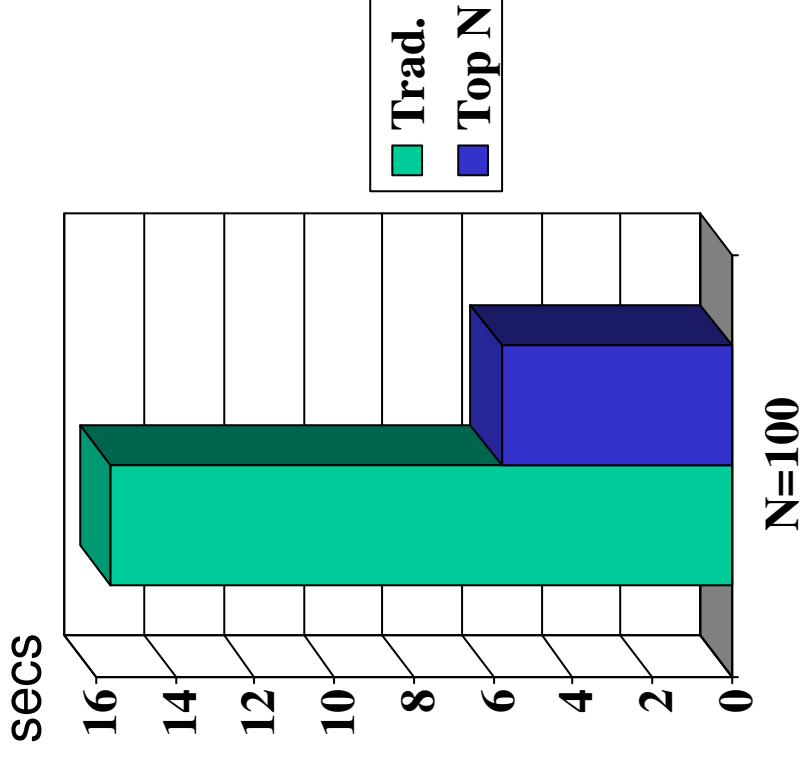
- traditional system does not use index and sorts Emps
- enhanced system uses index



Experiment 3: Cost of Sorting

100 highest paid Emps, DB2, no index, 10 MB DB

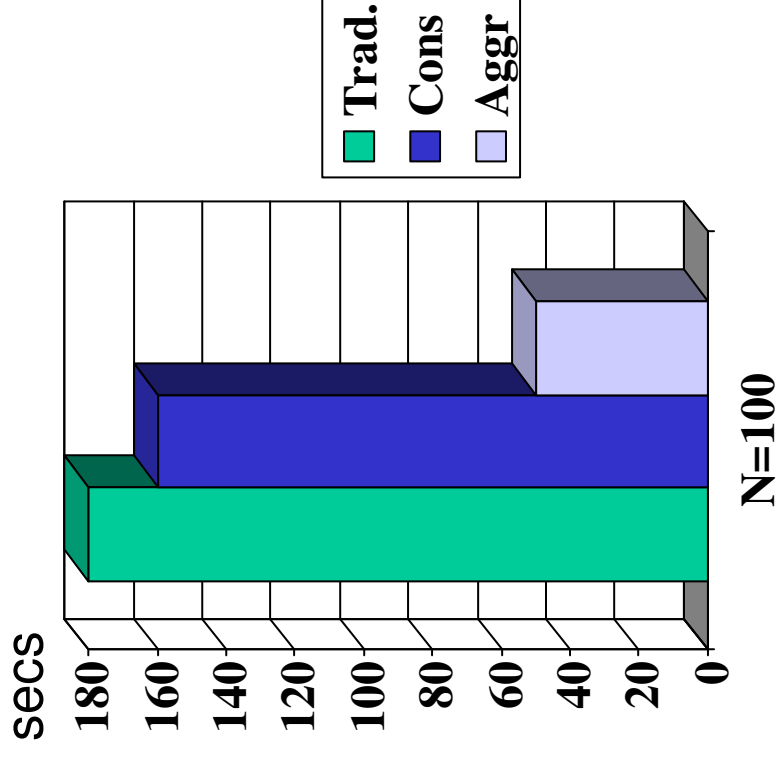
- traditional system sorts all Emps
- enhanced system uses priority queue



Experiment 4: Cons. vs. Aggr.

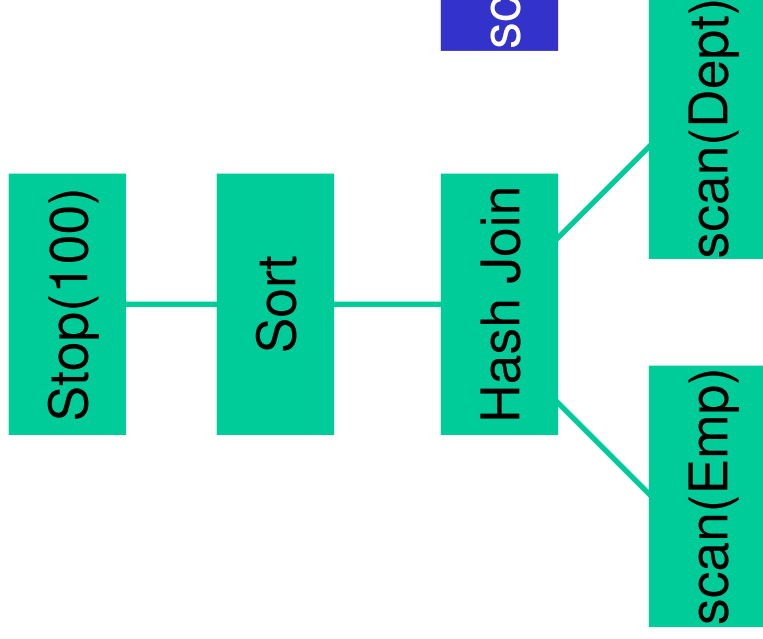
100 highest paid Emps and Dept, AODB, no index, 50 MB DB

- **traditional system** joins all Emps with all Depts, sorts whole result
- **Conservative:** joins all Emps with all Depts, uses PQ for top N
- **Aggressive:** joins a couple of hundred Emps with Depts (hybrid range partitioning)
- **Aggressive:** each restart costs ~30 secs

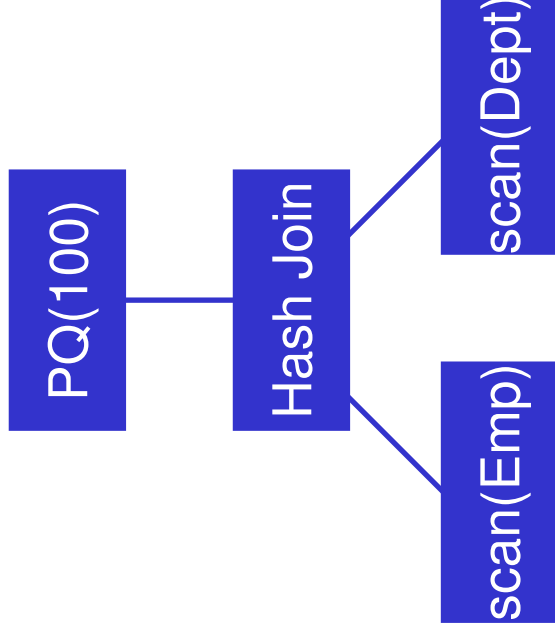


Experiment 4: Query Plans

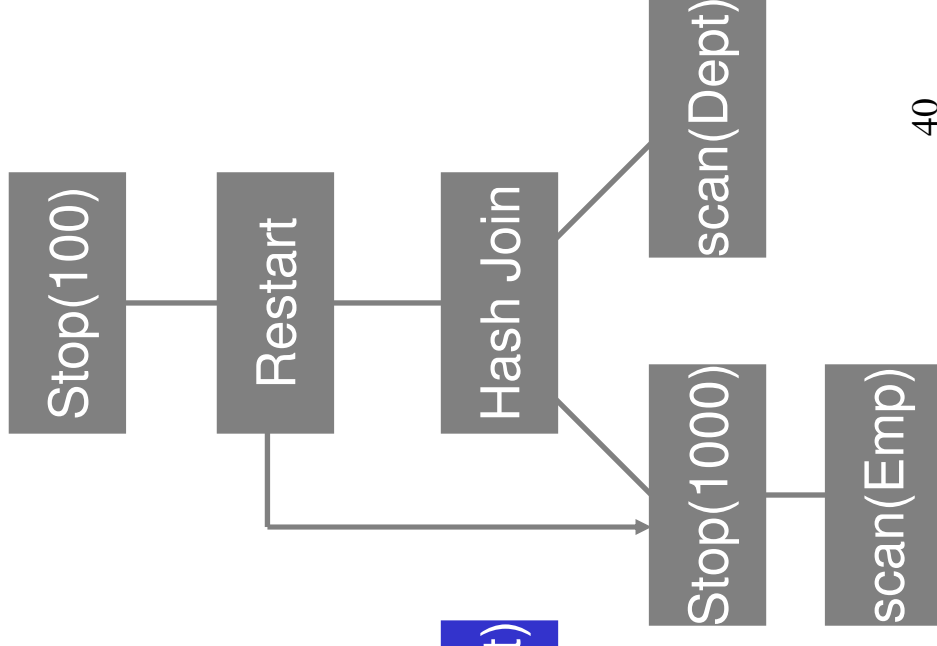
Traditional



Conservative



Aggressive



Summary: Experiments

- Top N needs to be integrated into DBMS
 - traditional (external) approach loses in all cases
 - differences are particularly high with indexes
- Range-partitioning very useful
 - executing a query several times with little data is often better than once with all the data
 - the more complex the query, the higher the gains
 - the more complex the query, the higher the risks

Related Work

- Related SQL language extensions
 - [Strehlo & Kimball 95], all major DBMS vendors
- Top N for multi-media data (pre-ordered data)
 - [Fagin 96, Chaudhuri & Gravano 96]
- Optimization for first few answers
 - [Bayardo & Miranker 96], ...
- Follow-up work on top N query processing
 - [Chaudhuri & Gravano 99, Ooi et al. 99, Donjerkovic & Ramakrishnan 99]

Conclusion

- STOP AFTER simple and useful SQL extension
- Top N query processing
 - several techniques combined give best performance
 - easy to parallelize
 - *next* operation easy to implement
 - **top N queries are faster than regular queries**
- Great impact
 - many applications
 - all major DB vendors are doing something about it

Future Work

- **Statistics; order statistics**
 - what are the right statistics to keep?
- **Risk-based, probabilistic query optimization**
 - [Donjerkovic & Ramakrishnan 99]
 - (related work by [Seshadri 99])
- **Multi-dimensional variants of top N**
 - implement and evaluate Skyline operator
 - investigate other models
- **Range-partition for other types of queries**