Squall: Scalable Real-time Analytics

Aleksandar Vitorovic

EPFL, Switzerland

Motivation

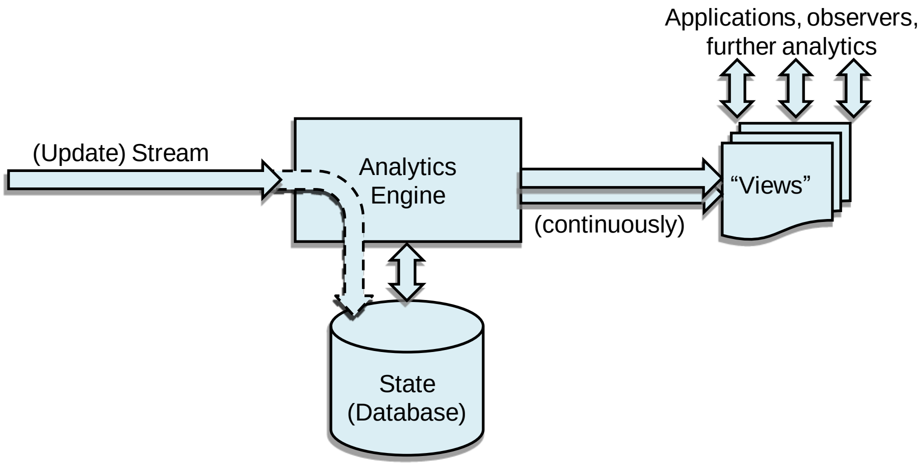
- Processing large data in a scalable way
  - Terabytes of logs (e.g., Loggly) and clickstreams
  - Processing large scientific data (e.g., LHC)
  - Exploratory queries on scientific data
- Real-time surveillance, traffic and infrastructure monitoring
  - Scheduling algorithm (over Google Cluster Data)
- Business intelligence
  - Finding patterns in customer and sales data
  - Online advertising: QuantCast
  - Reach a potential customer during the active session
- Online anomaly and fraud detection
- Real time virtual auctions systems: Ebay, algorithmic trading

Outline

- Motivation
- Squall: Scalable Real-time Analytics
- Resource-aware query optimization
- Efficient operators in Squall
- Skew-resilient partitioning schemes for
  - 2-way joins
  - Multi-way joins
- Local operators
- Skew in online systems
- Conclusion

Online processing

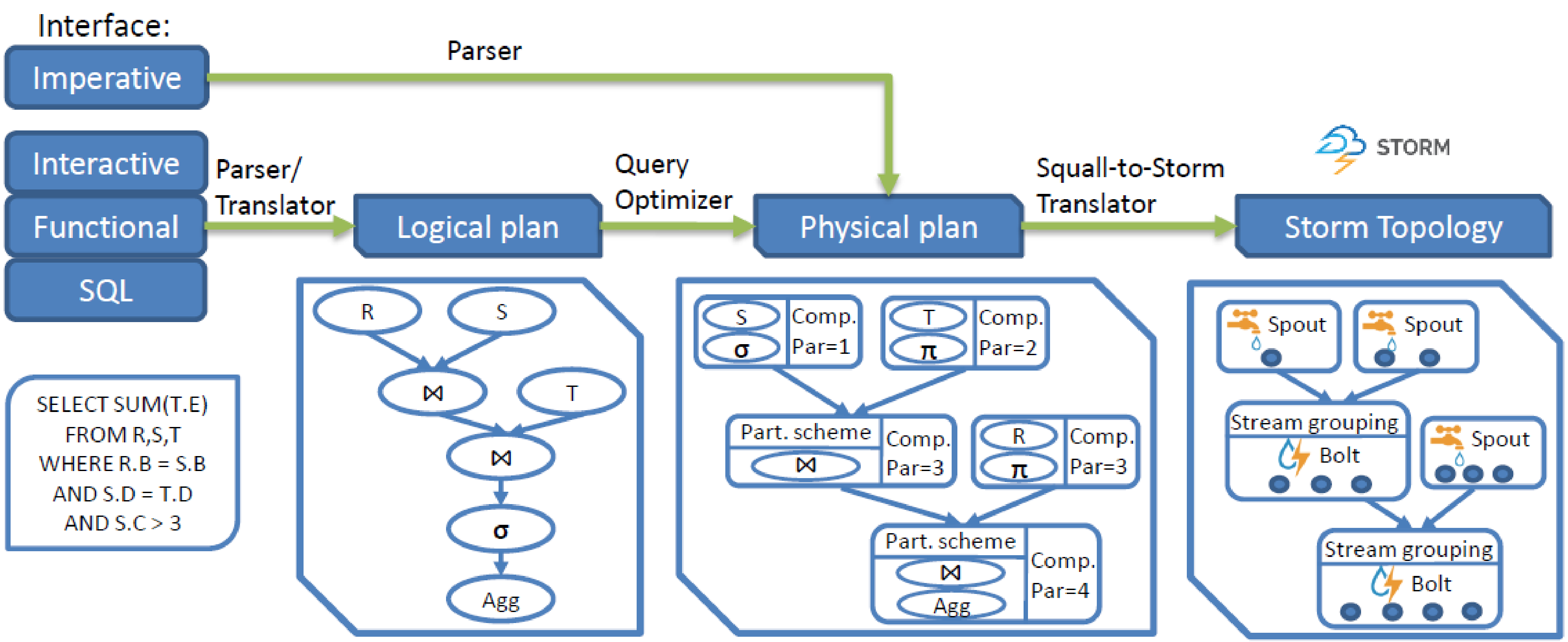
- Results are incrementally built as the input arrives
- Each input tuple produces output and updates the system state necessary for processing subsequent inputs
- Semantics
  - Full-history semantics (Incremental View Maintenance)
  - Window (stream) semantics



Offline (batch) vs online (real-time) processing

Scalable Analytics

- Complex queries with operators:
  - Selections, projections, joins and aggregations
- Distributed setting: shared-nothing architecture
- Scalable: high input rates and/or high memory requirements
  - Each operator runs in parallel
  - Operator: partitioning scheme and local join operator

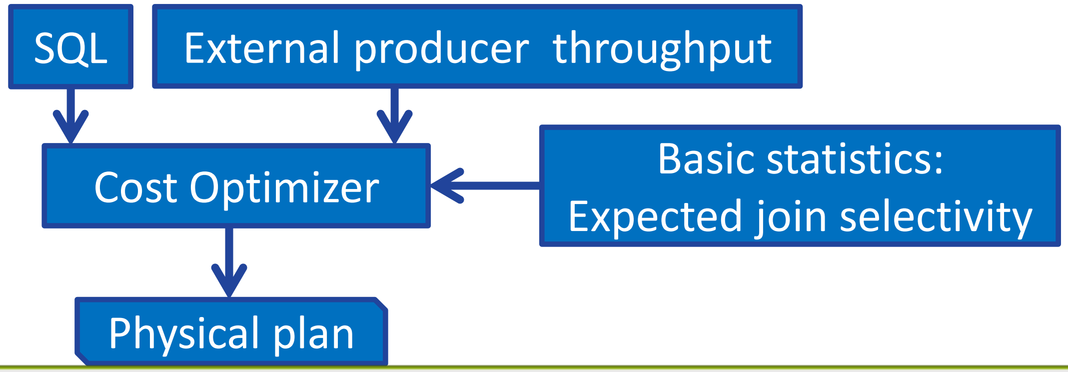


Outline

- Motivation
- Squall: Scalable Real-time Analytics
- Resource-aware query optimization
- Efficient operators in Squall
- Skew-resilient partitioning schemes for
  - 2-way joins
  - Multi-way joins
- Local operators
- Skew in online systems
- Conclusion

Resource-aware query optimizer

- Squall translates SQL to a distributed query plan
  - Hive extends Hadoop, Squall extends Storm
- Goal: maximize throughput, minimize latency and # of nodes
- Key ideas:
  - Universal Producer-Consumer balance
    - Parallelism(Comp) = f(throughput(Upstream), operation(Comp))
    - Sweet spot between latency and node utilization
  - Dynamic programming: # of nodes as the only objective

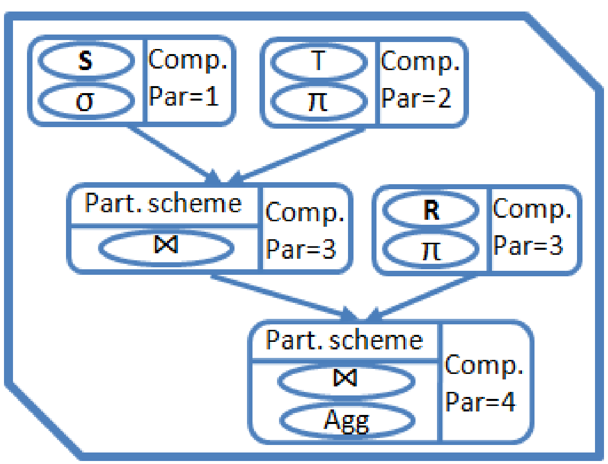


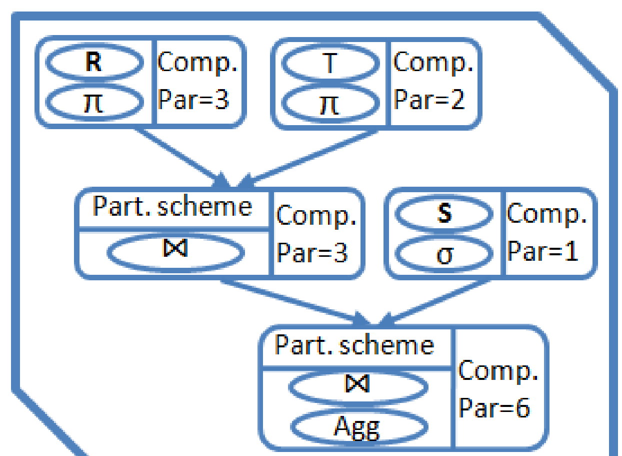
Translate to a 1-objective problem; use dynamic prog.

Dynamic programming

- Build plans starting from data sources, and keep only the best query plan for each subplan
  - Parallelism: not overloaded, not mostly idle
  - Co-locate operators to components to minimize network transfers
  - Pushing up selections and projections
  - CSE (Common subexpression elimination)

- Example: Best (sub-)plan for R ⋈ T ⋈ S
  - # nodes((S ⋈ T) ⋈ R) < # nodes((R ⋈ T) ⋈ S)



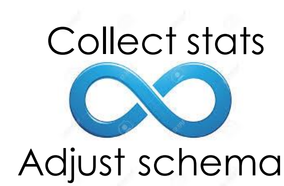


Outline

- Motivation
- Squall: Scalable Real-time Analytics
- Resource-aware query optimization
- Efficient operators in Squall
- Skew-resilient partitioning schemes for
  - 2-way joins
  - Multi-way joins
- Local operators
- Skew in online systems
- Conclusion

Squall puts together:

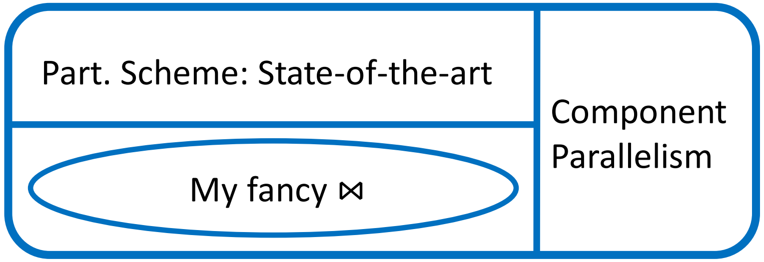
- Skew-resilient partitioning schemes
  - Existing open-source online distributed systems (Storm, Spark Streaming, Flink) has only vanilla database operators
  - Squall provides novel skew-resilient schemes for 2-way/multi-way joins
  - Choose according to data distribution (skew)
- Local query operators
  - Hash and Balanced Binary Tree Indexes
  - DBToaster
  - Choose according to # of relations, join conditions
- Techniques for scalable online query processing
  - Adaptive operators



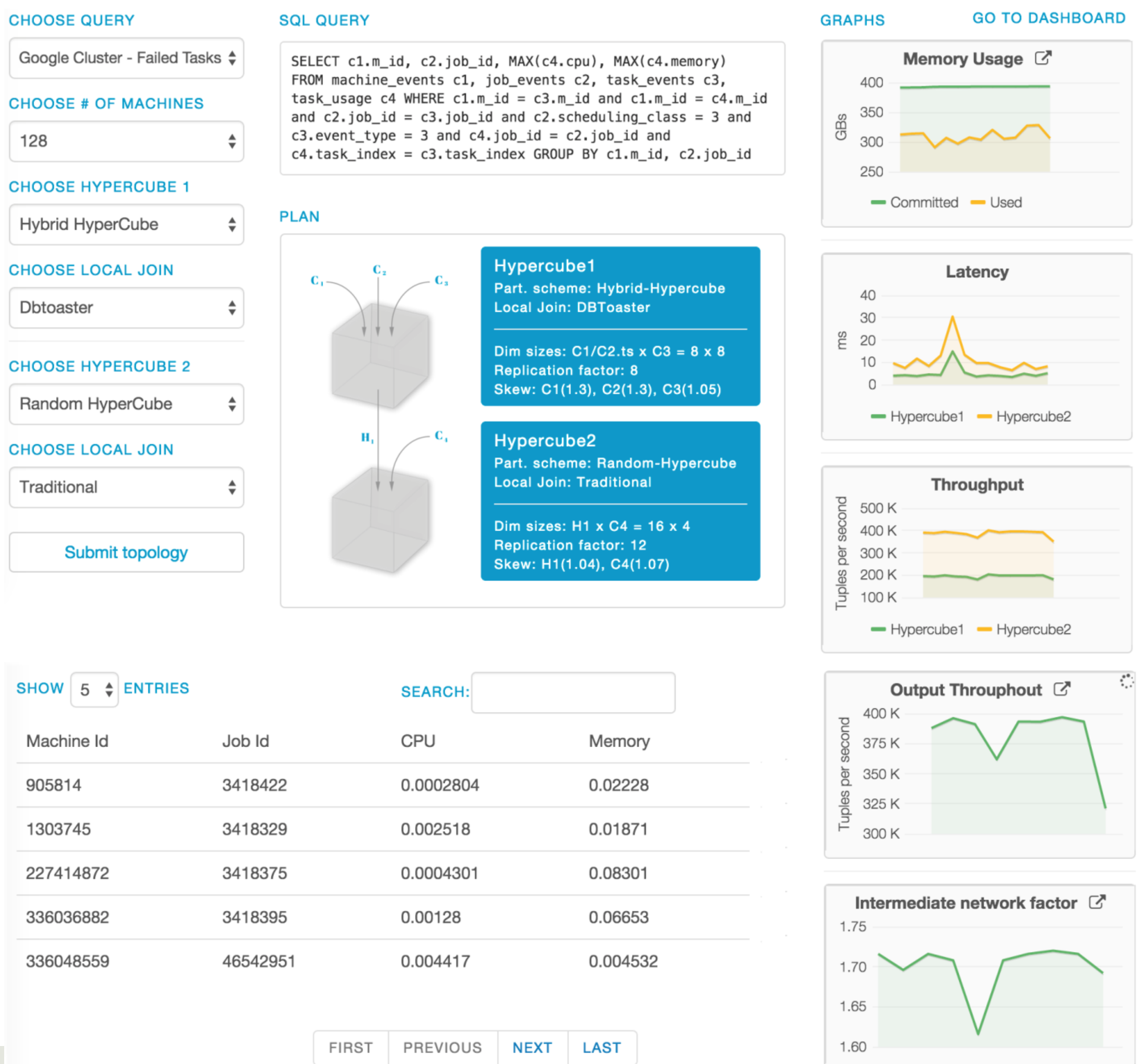
Squall operators are skew resilient, adaptive and scalable

Modular design

- Leverage the effect of various design choices on the performance
  - Different operators have different characteristics depending on: # of relations, join conditions, data distribution (skew)
- Seamlessly build efficient novel operators
  - Combine a partitioning scheme with a local operator
- Discover and address new skew types that arise only in online systems



Web interface



Squall is not just a toy

- https://github.com/epfldata/squall
- It has been developed for the last five years
  - Mainly by the authors at EPFL, but also with external contributions
- Squall has attracted a community of users
- Twitter: They chose us among multiple different options
- Accepted papers:
  - *Squall: Scalable Real-time Analytics.* **A. Vitorovic**, M. ElSeidy, K. Guliyev, K. Vu, D. Espino, M. Dashti, Y. Klonatos and C. Koch. VLDB Demo 2016
  - *Load Balancing and Skew Resilience for Parallel Joins.* **A. Vitorovic**, M. ElSeidy and C. Koch. ICDE 2016
  - *Scalable and Adaptive Online Joins.* M. ElSeidy, A. Elguindy, **A. Vitorovic** and C. Koch. VLDB 2014

Outline

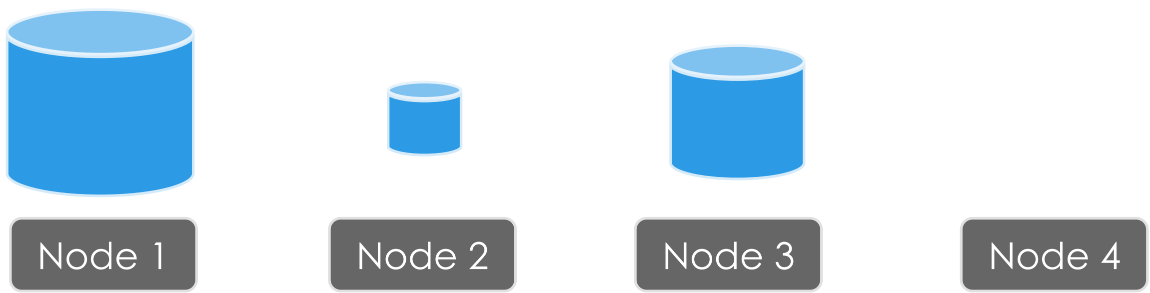
- Motivation
- Squall: Scalable Real-time Analytics
- Resource-aware query optimization
- Efficient operators in Squall
- Skew-resilient partitioning schemes for
  - 2-way joins
  - Multi-way joins
- Local operators
- Skew in online systems
- Conclusion

2-way join: Analytics go beyond equi-joins

- Examples of inequality/band joins
  - BI: Advertise a product only to users who can afford it
  - Time- and space-distance joins (locating nearby events or objects)
    - Call logs analytics (e.g., base station misconfiguration)
    - Weather station analytics (e.g., storm propagation)
    - Astronomy (e.g., sky configuration)
    - Online geospatial analysis: Waze
- Our focus
  - Monotonic joins: Combinations of equi- and inequality joins
    - Band join is a combination of 2 Inequality joins
  - Low-selectivity joins

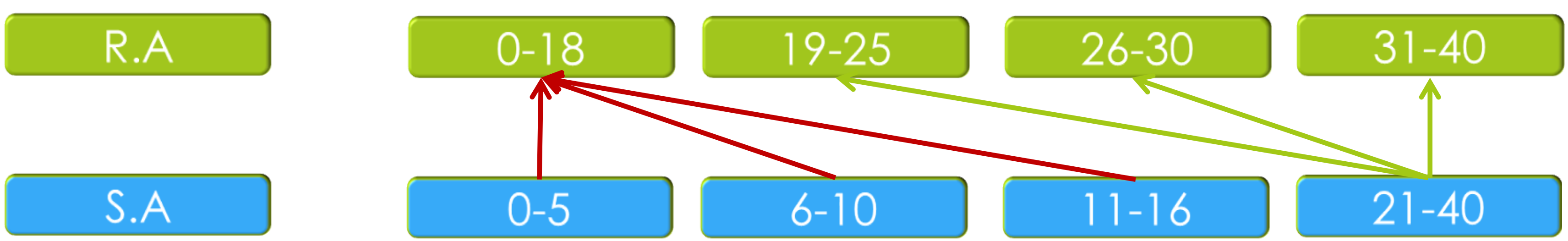
Distributed Setting

- Hash key partitioning



1. Only = joins
2. Limited parallelism
3. Vulnerable to skew

- Range partitioning using data distribution of one relation:
  - Join condition: `|R.A - S.A| <= 2`



Skew remains: 3X more load for one R partition

Skew

- Skew (e.g., zipfian distribution) occurs frequently:
  - Internet packet traces
  - Word frequency in natural languages
- Skew types
  - Redistribution skew (RS): uneven input data partitioning among the machines due to skew in the join keys
  - Join product skew (JPS): imbalance in # of produced output tuples due to variability in the join selectivity
  - A small number of machines process most of the data

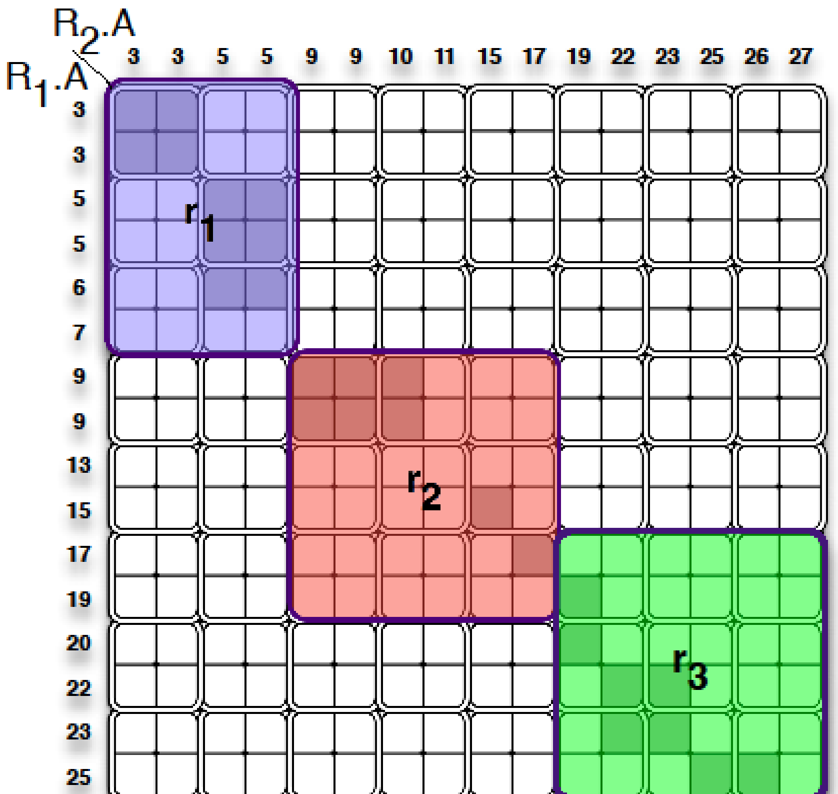
A good partitioning scheme addresses both skew types

Join Matrix model

- Models join as Cartesian space, range partitioning on both relations
- Join-matrix M: M(i,j) = *true*, iff ($r1_i$, $r2_j$) is in join result
- Partitions into independent regions → assign to machines

$|R_1.A - R_2.A| <= 1$

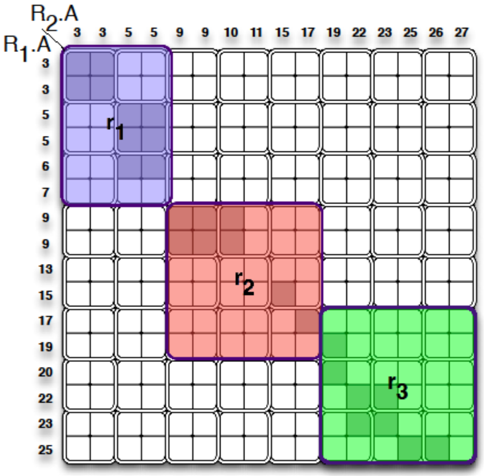
R1. A	R2. A	Output tuple
25	25	true
25	26	true
3	3	true
…		



Range partitioning → Discard large matrix portions

Load-balancing optimization goal

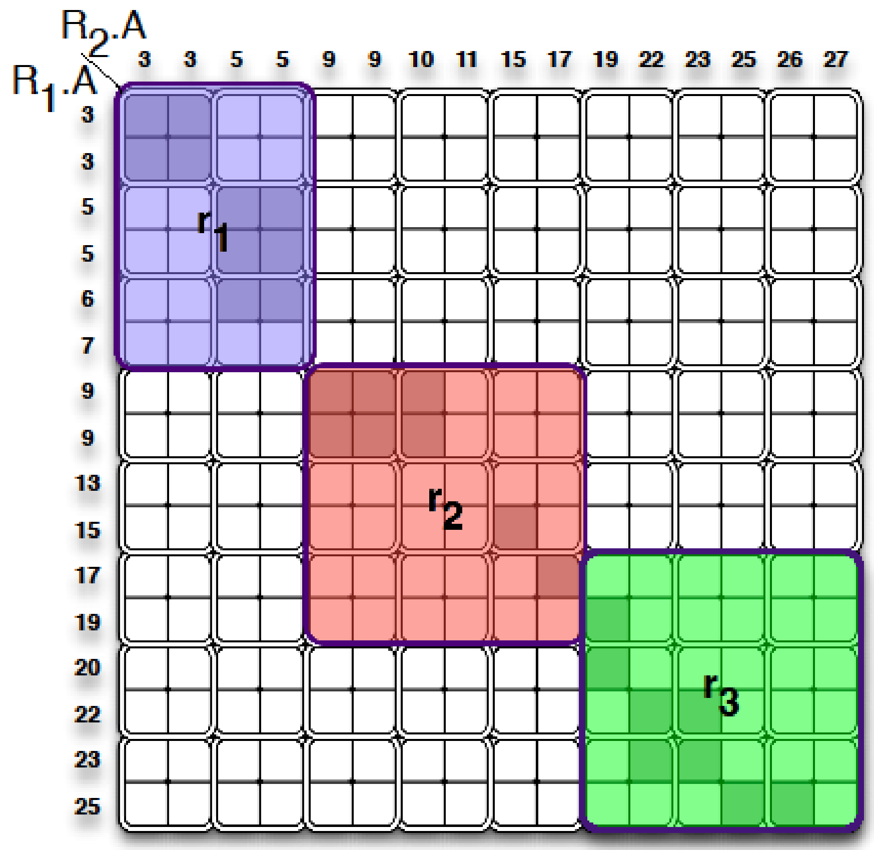
- Minimize the maximum join work per node (W_{MAX})
- W_{MAX} = weight(region) = function(# of inputs, # of outputs)
- # of input tuples (semi-perimeter) of a region
  - Demarshalling the tuple/performing the join
  - Includes communication and storage costs
- # of output tuples produced in a region
  - A post-processing stage
    - Writing to disk, or
    - Transferring over the network for later stages



Region weight corresponds to machine work

Towards equi-weight histograms

- W_{MAX} = weight(region) = function(# of inputs, # of outputs)
- Example: weight (region) = # of inputs + # of outputs
- $w_1 = 10 + 10 = 20$
- $w_2 = 12 + 8 = 20$
- $w_3 = 12 + 8 = 20$



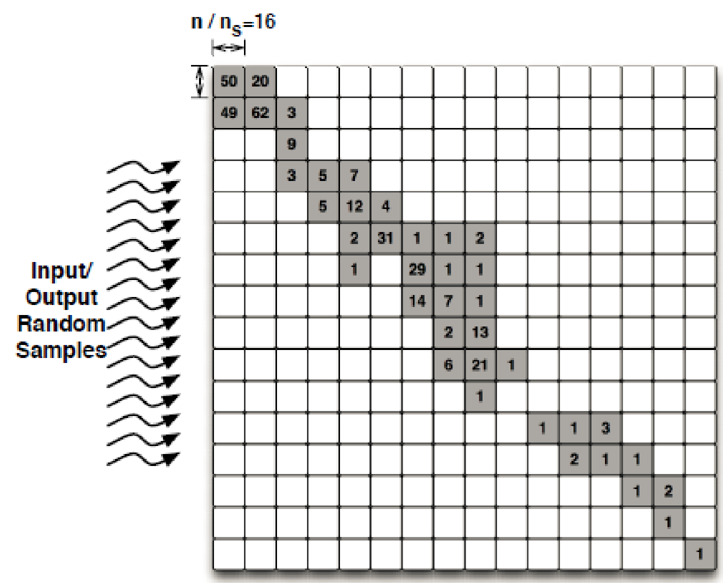
Build an equi-weight histogram

- Rectangle tiling problem in Computational geometry
  - Algorithm: Binary Space Partitioning (BSP)
- BSP takes $O(n^5 \log n)$ time
  - More expensive than the join itself
- We design a multi-stage algorithm that runs in $O(n)$ time:
  - Reduce the input size for BSP
  - Devise a join-specialized BSP

Applying existing algorithms is infeasible

Reducing input size – part I

- Employ sampling
  - Run BSP on the Sample, rather than the original matrix
  - Sample matrix M_S preserves the input/output cost distribution from the original join matrix
- Capture the input distribution:
  - Equi-depth histogram on each relation
- Capturing the output distribution
  - Without performing the entire join
  - We devise a parallel Stream-Sample
    - *On Random Sampling over Joins*, Chaudhuri et al., *SIGMOD '99*



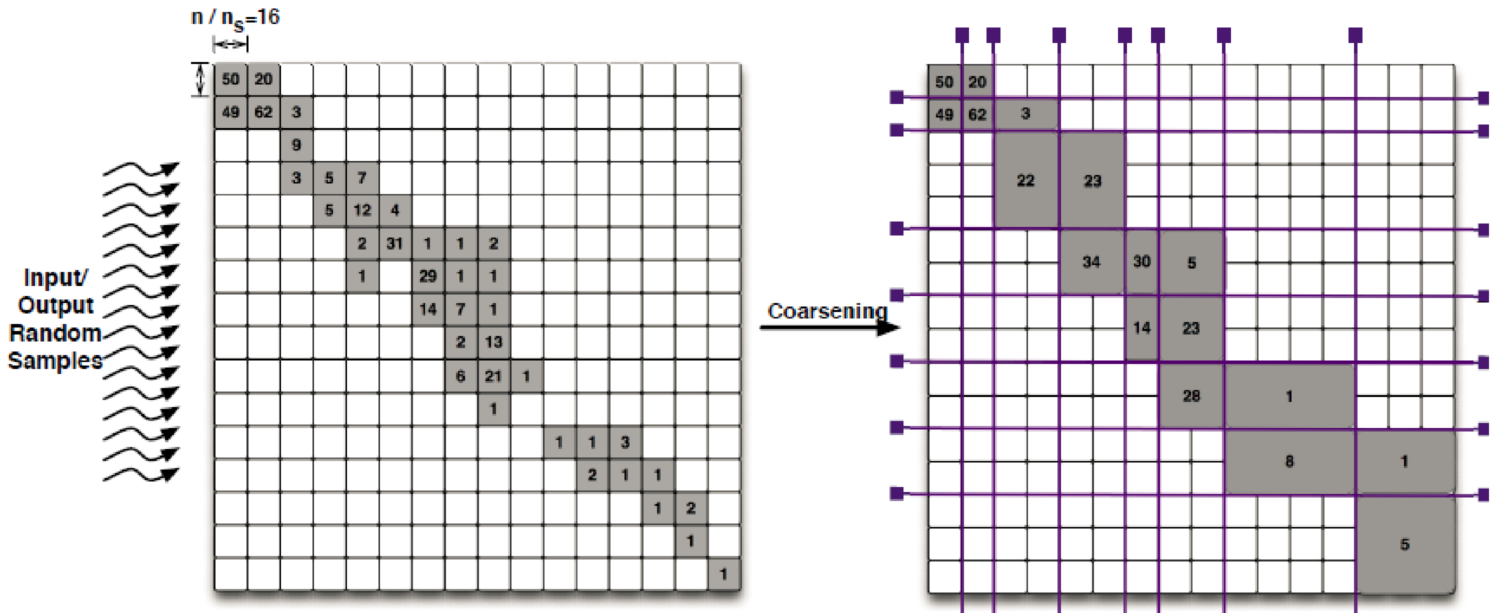
a) Sample Matrix M_S
n_S x n_S = 16 x 16

- BSP over sample matrix M_S: $O((nJ)^{2.5}\log n)$ time

Still too expensive!

Reducing input size – part II

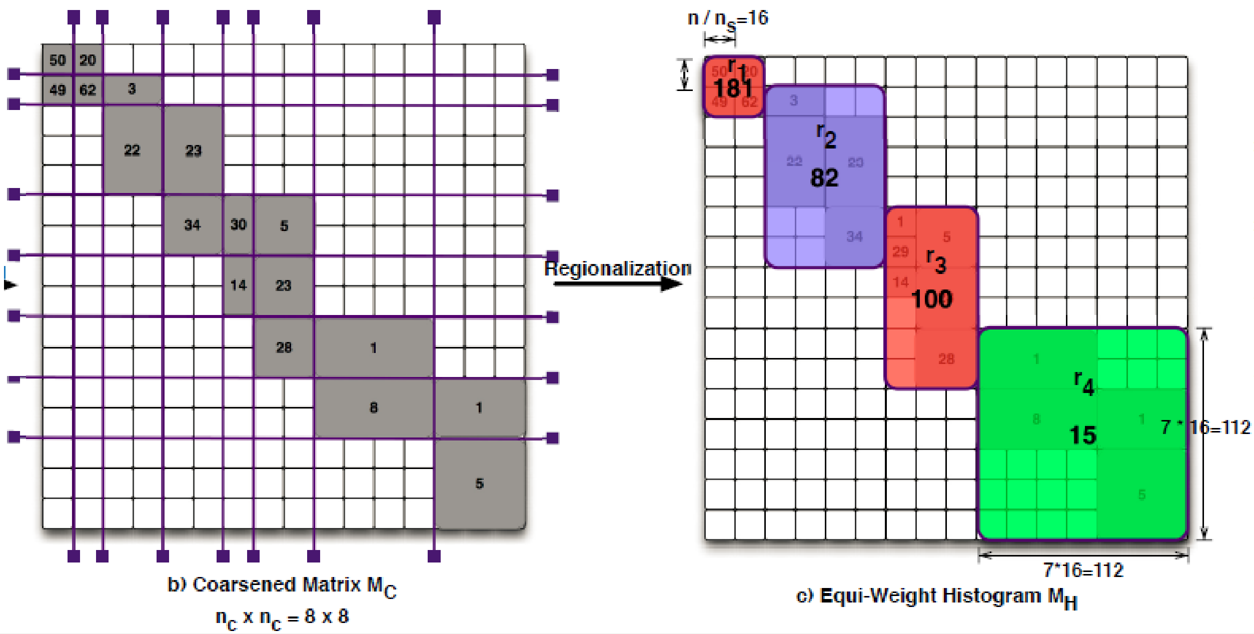
- Employ coarsening
  - Reduce the BSP input by creating a non-uniform grid over sample matrix
  - The goal is to minimize the maximum cell weight in the coarsened matrix
  - We represent multiple small M_S cells as one M_C cell
- BSP over coarsened matrix M_C: $O(n^{5/3} \log n)$ time



Still too expensive!

MonotonicBSP

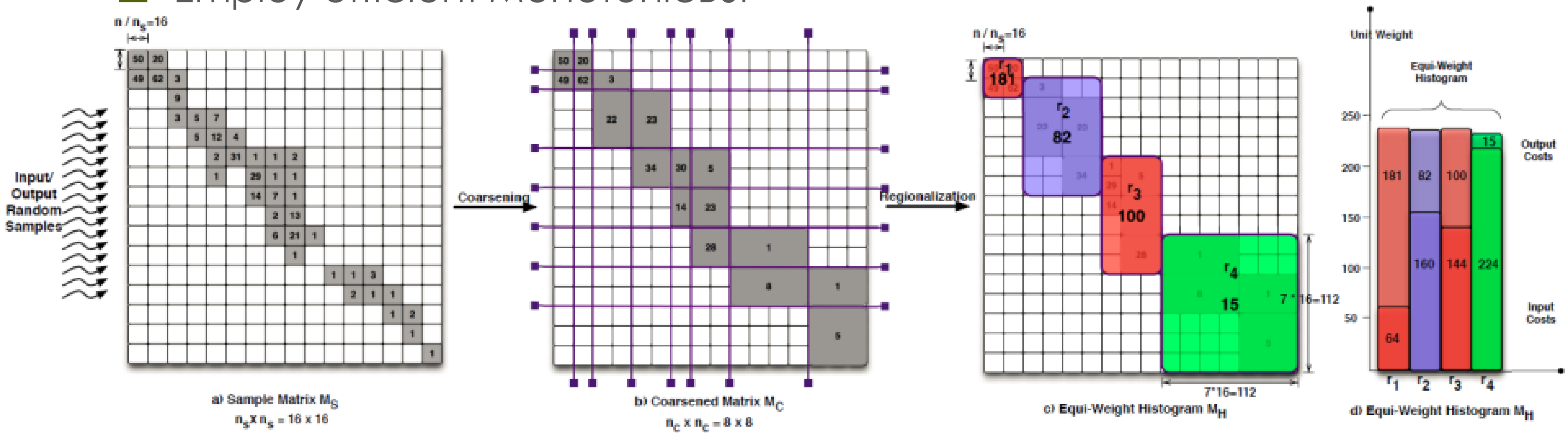
- A novel, join-specialized tiling algorithm
  - Exploits *properties of the join output distribution* for equi-joins, band-joins and inequality-joins
  - MonotonicBSP provides the same output as the original BSP
- MonotonicBSP over coarsened matrix M_C takes only $O(n)$ time!



Running time finally acceptable!

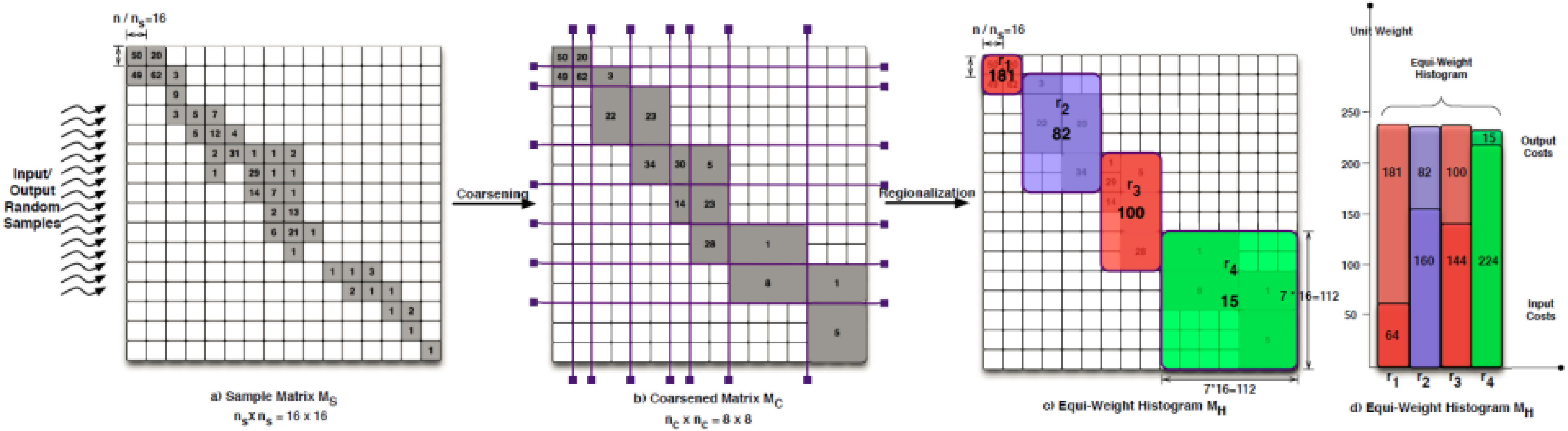
Putting everything together

- Careful choice of sample and coarsened matrix sizes n_S and n_C
  - $n > n_S > n_C$
  - n_S and n_C small enough to keep the running time short
  - Insufficient stage output granularity leads to inaccurate load balancing
    - One machine is assigned more work than others
- Efficiency of our multi-stage algorithm
  - Reduce the input size for the BSP
  - Employ efficient MonotonicBSP



Putting everything together

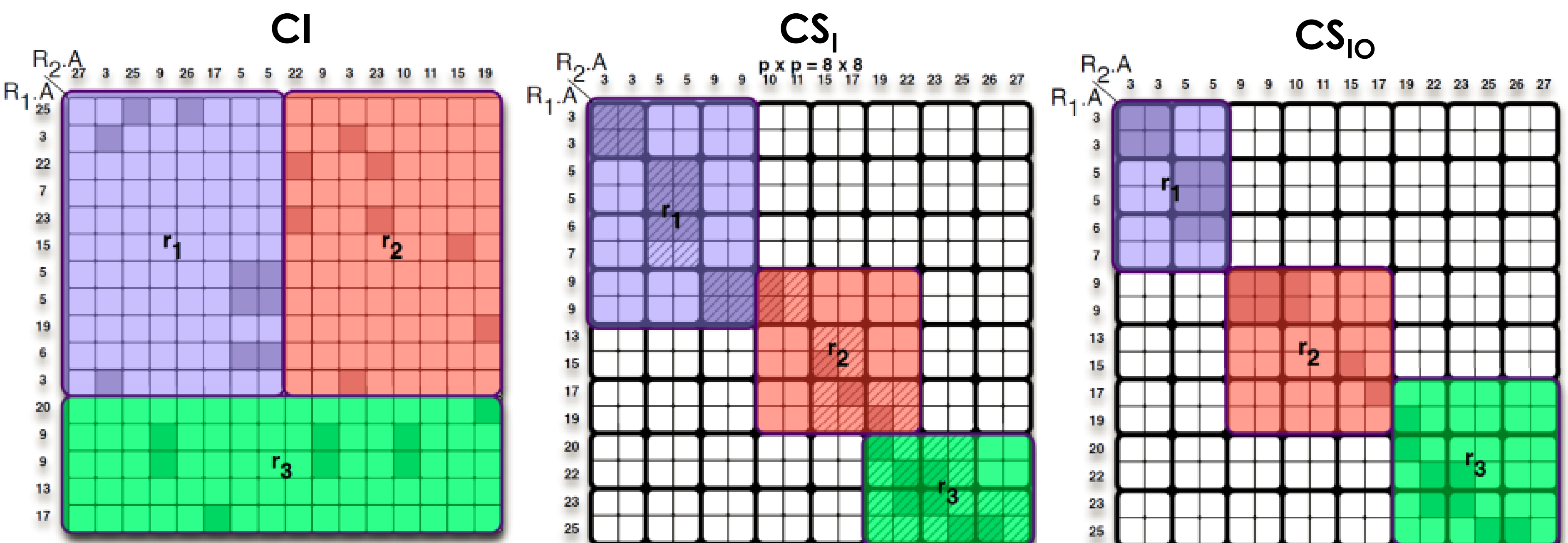
- Accuracy
  - We avoid imposing any assumptions about distribution within a cell
    - Each stage coalesces multiple cells from the previous stage
  - More precise algorithms for more coarse-grained input
    - Sampling: uniform
    - Coarsening: non-uniform grid
    - BSP: hierarchical partitioning



Careful algorithm design + matrix sizes: Efficiency + Accuracy

State-of-the art join operators

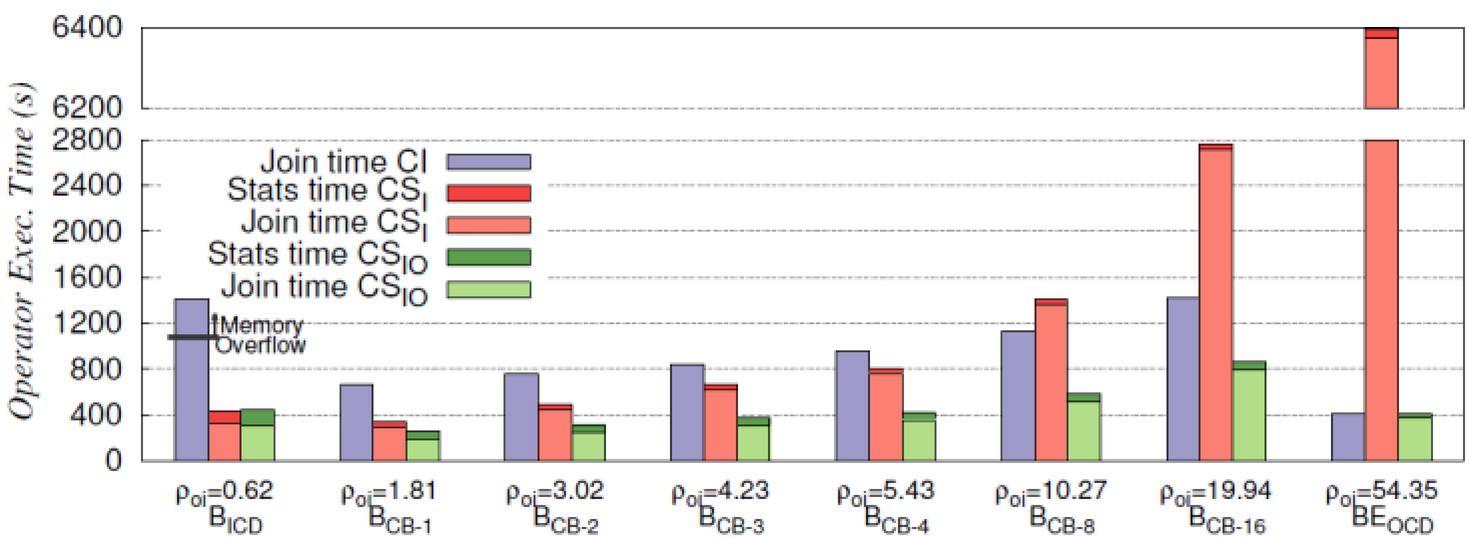
- **CI**: randomly assigns join keys to rows/columns
  - +: Perfect load balancing of output
  - -: Excessive input-related work (each cell is assigned to a machine)
- **CS_I**: range partitioning, heuristic algorithm for partitioning
  - +: Acceptable input-related cost
  - -: Prone to JPS (does not capture the output distribution)
- **CS_{IO}** : our scheme achieves the best from both worlds



Results

- Only our CS_{IO} performs well over a wide range of output/input sizes
  - CI performs well only for input-dominated joins
  - CS_I performs well only for output-cost dominated joins
- Problem: Hard to know the output/input beforehand
  - Output-size estimation techniques are error-prone (Ioannidis1991)
  - We are dealing with non-equi joins

- 160G z1 TPCH data
- Cluster of 32 VMs
- 3GHz Xeon
-1Gbit Ethernet

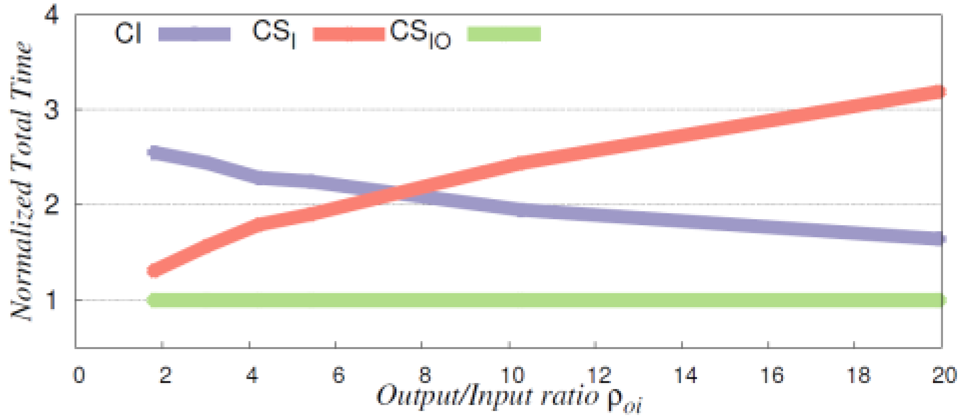


Our scheme achieves up to 15X speedup!

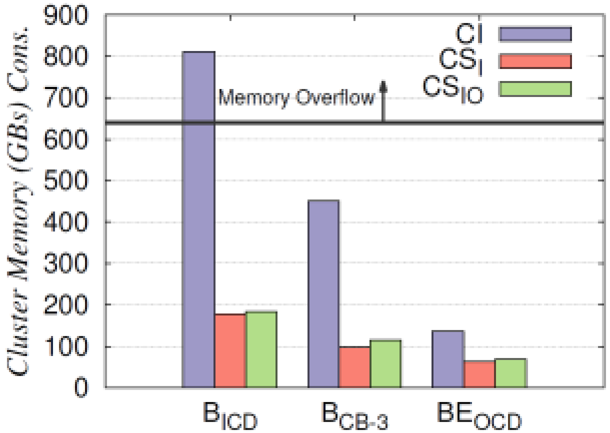
Results

- Only our scheme performs well over a wide range of output/input sizes

Bcb



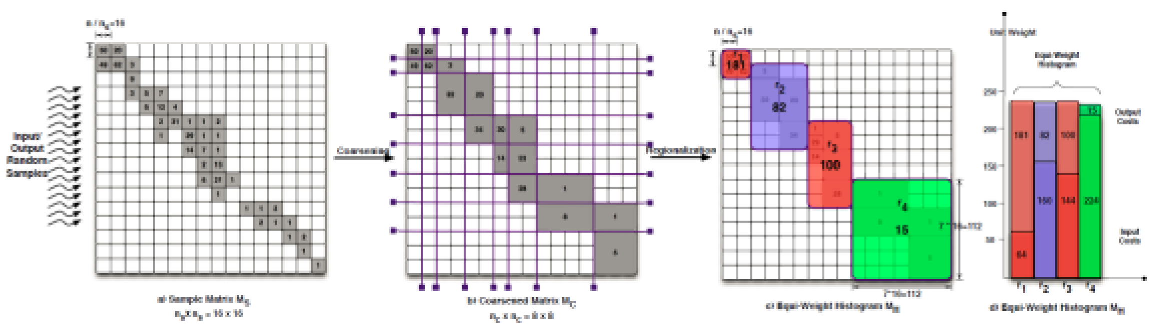
- Our scheme achieves up to 5X improvement in resource consumption



Various data distributions & resource efficiency

Recap

- Efficient and accurate load balancing for monotonic joins
  - Minimize the maximum work per machine
  - With no assumptions on the input or output sizes, or data distribution
- Multi-stage algorithm
  with a join-specialized computational geometry algorithm
- Experimental results
  - Up to 15X speedup compared to state-of-the-art
  - Up to 5X more efficient in terms of resource utilization



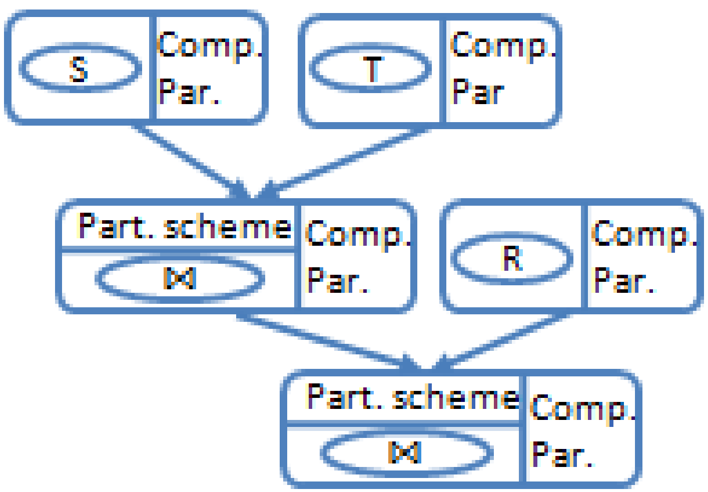
Outline

- Motivation
- Squall: Scalable Real-time Analytics
- Resource-aware query optimization
- Efficient operators in Squall
- Skew-resilient partitioning schemes for
  - 2-way joins
  - Multi-way joins
- Local operators
- Skew in online systems
- Conclusion

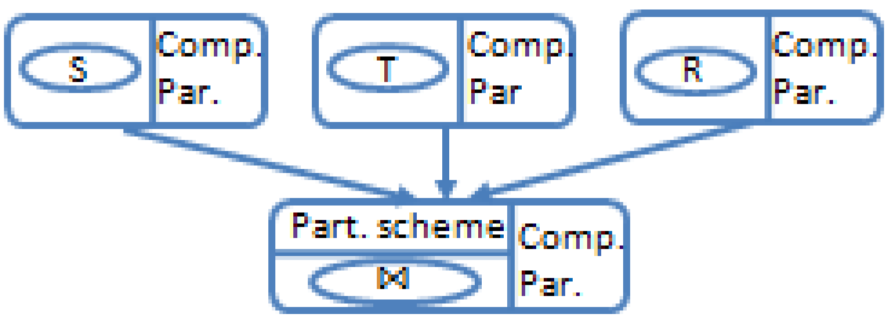
Multi-way joins

- Multi-way join implies single communication step (component)
- Motivation
  - Intermediate relations are large (e.g., Reachability query)
  - Online: No need to change join orders (as for pipeline of 2-way joins)

Pipeline of 2-way joins

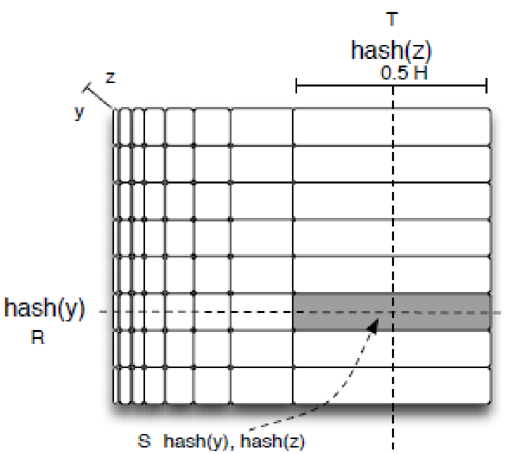


Multi-way join

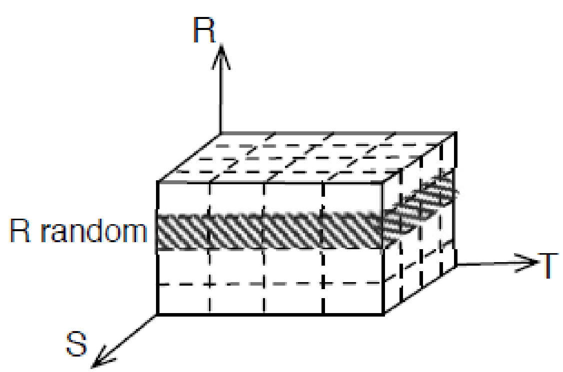


Multi-way joins

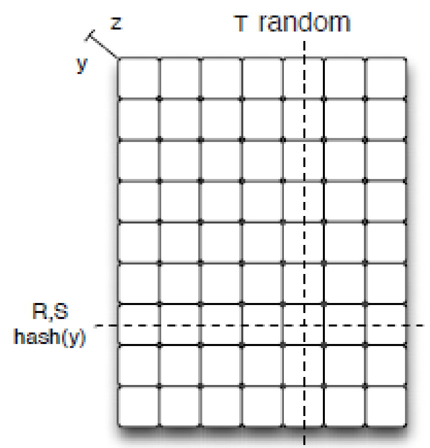
- Hash-Hypercube: hash partitioning with attributes as hypercube dim.
  - Skew-free multi-way equi-joins
- Random-Hypercube: random partitioning with relations as dim.
  - High-replication: 3-dimensional rather than 2-dimensional hypercube
- Hybrid-Hypercube: hash partitioning (for skew-free join keys); random partitioning (in the case of skew or non-equi joins)
  - Load per machine is 2.08X and 1.92X smaller than for the existing schemes



Hash-Hypercube with skew.



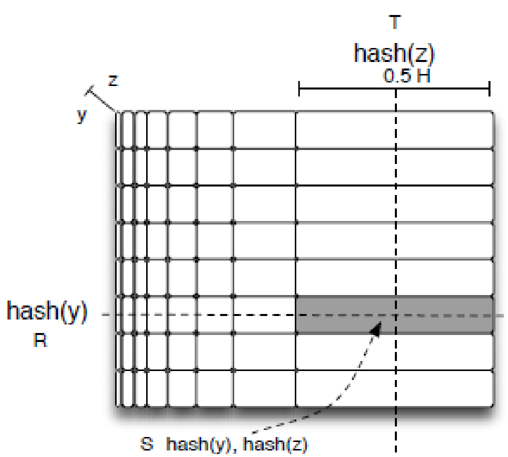
Random-Hypercube.



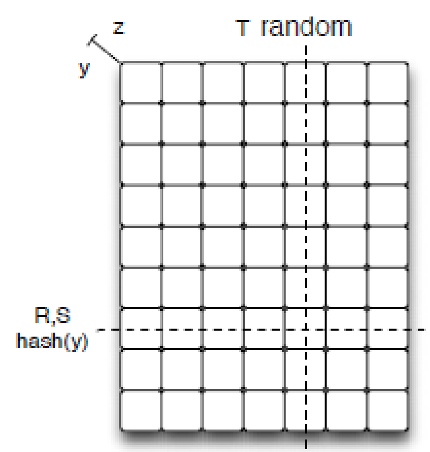
Hybrid-Hypercube.

Multi-way joins: Optimization algorithm

- How to choose dimension sizes?
- Hash-Hypercube: $L = \frac{R}{p_y} + \frac{S}{p_y p_z} + \frac{T}{p_z}$ $\quad p_y p_z = 64$
- Hybrid-Hypercube
  - Hard: Variety of join conditions and different data distributions
  - R.y, S.y: hash partitioning; S.z, T.z: random partitioning
  - Rename S.z to z' $\quad L = \frac{R}{p_y} + \frac{S}{p_y p_{z'}} + \frac{T}{p_z}$ $\quad p_{z'} = 1$ $\quad L = \frac{R+S}{p_y} + \frac{T}{p_z}$



Hash-Hypercube with skew.



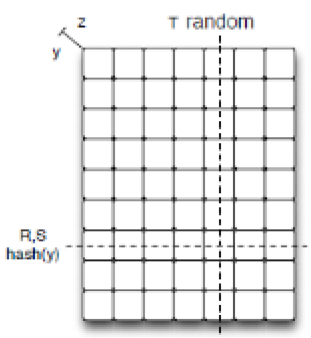
Hybrid-Hypercube.

Outline

- Motivation
- Squall: Scalable Real-time Analytics
- Resource-aware query optimization
- Efficient operators in Squall
- Skew-resilient partitioning schemes for
  - 2-way joins
  - Multi-way joins
- Local operators
- Skew in online systems
- Conclusion

Local operators

- Efficient in-memory data-structures
  - Hash and Balanced Binary Tree Indexes
  - Collections of primitive types
  - Byte Arrays for Classes (e.g., Strings)
- DBToaster, a very efficient local join
  - Materializes intermediate relations and reuses them
  - Orders of magnitude performance improvement for multi-way joins
  - DBToaster was previously considered hard to parallelize



Hybrid-Hypercube.

+ TOASTER = **HyLD operator**

Network efficiency **CPU efficiency**

Outline

- Motivation
- Squall: Scalable Real-time Analytics
- Resource-aware query optimization
- Efficient operators in Squall
- Skew-resilient partitioning schemes for
  - 2-way joins
  - Multi-way joins
- Local operators
- Skew in online systems
- Conclusion

Skew in online systems

- Skew fluctuations
  - Example: skew and the most popular key changes over time
  - Range partitioning: adjust the scheme (state migration over the network)
    - What if distribution changes right after the repartitioning?
  - Random partitioning: no need to adjust
- Temporal skew: tuples arrive in a specific order
  - Having uniform or the exact “offline” distribution may not suffice!
  - Example: tuples arrive in a sorted order
  - Range partitioning: only one machine active at a time (sequential exec.)
  - Random partitioning does not have this problem
- Join selectivity fluctuations: multi-way joins address this problem

Tradeoff between replication and skew-resilience

Outline

- Motivation
- Squall: Scalable Real-time Analytics
- Resource-aware query optimization
- Efficient operators in Squall
- Skew-resilient partitioning schemes for
  - 2-way joins
  - Multi-way joins
- Local operators
- Skew in online systems
- Conclusion

Conclusion

- Squall: a distributed online query engine
- Partitioning schemes: cover the spectrum of different data distributions:
  - 2-way joins
    - No assumptions on the join input or output sizes
    - A multi-stage alg./join-specialized computational geometry alg.
  - Multi-way joins
    - Do not require that either all or none of the relations is skew-free
    - A composite of different partitioning schemes according to the skew degree in different relation attributes
- Efficient local operators:
  - Efficient in-memory data structures & DBToaster
- Leverage the effect of various design choices on the performance:
  - Combine schemes and local operators using # of relations, join cond., skew
  - New skew types that arise only in online systems

Thank you!

Try SQUALL! Visit us at https://github.com/epfldata/squall