Mutant Query Plans

Vassilis Papadimos

vpapad@cs.pdx.edu

Portland State University

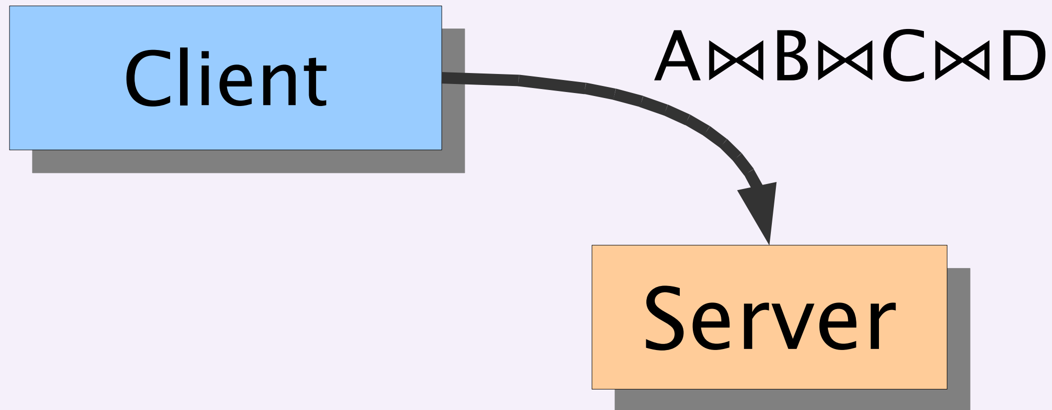
Scalable distributed querying

We have successful protocols and applications for distributed systems with millions of servers and users

- HTTP
- P2P filesharing

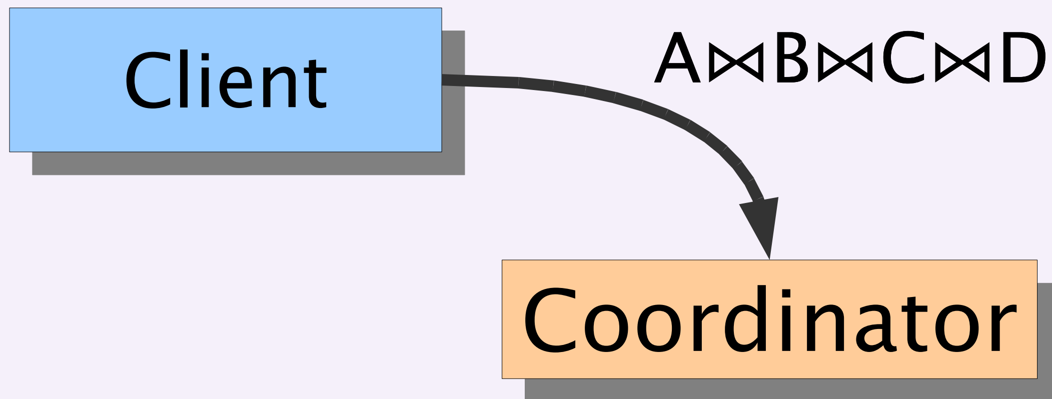
How do we build a system with a similar scale that supports complex declarative queries over distributed data?

The textbook approach



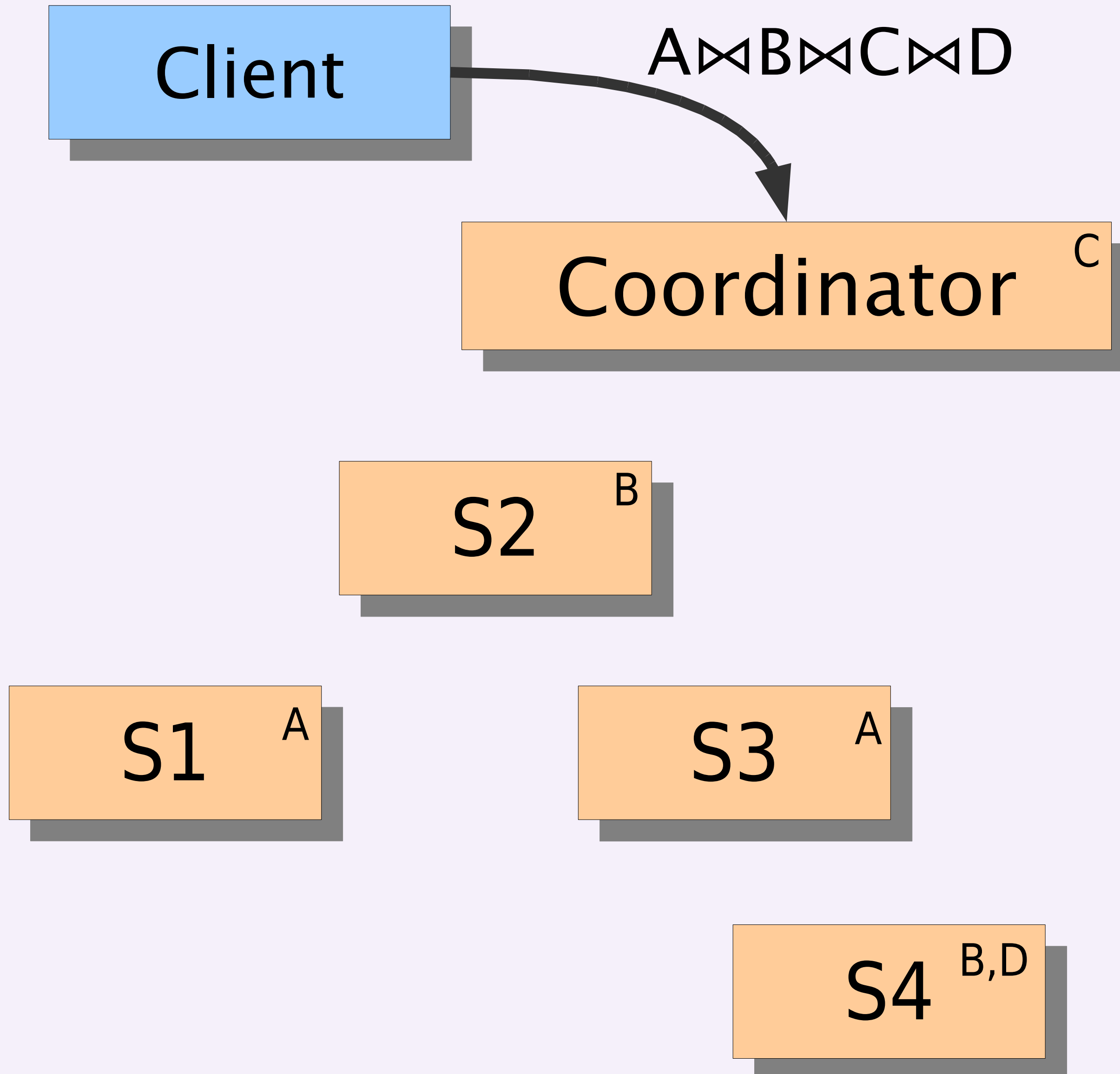
1. Connect to server, submit a query

The textbook approach



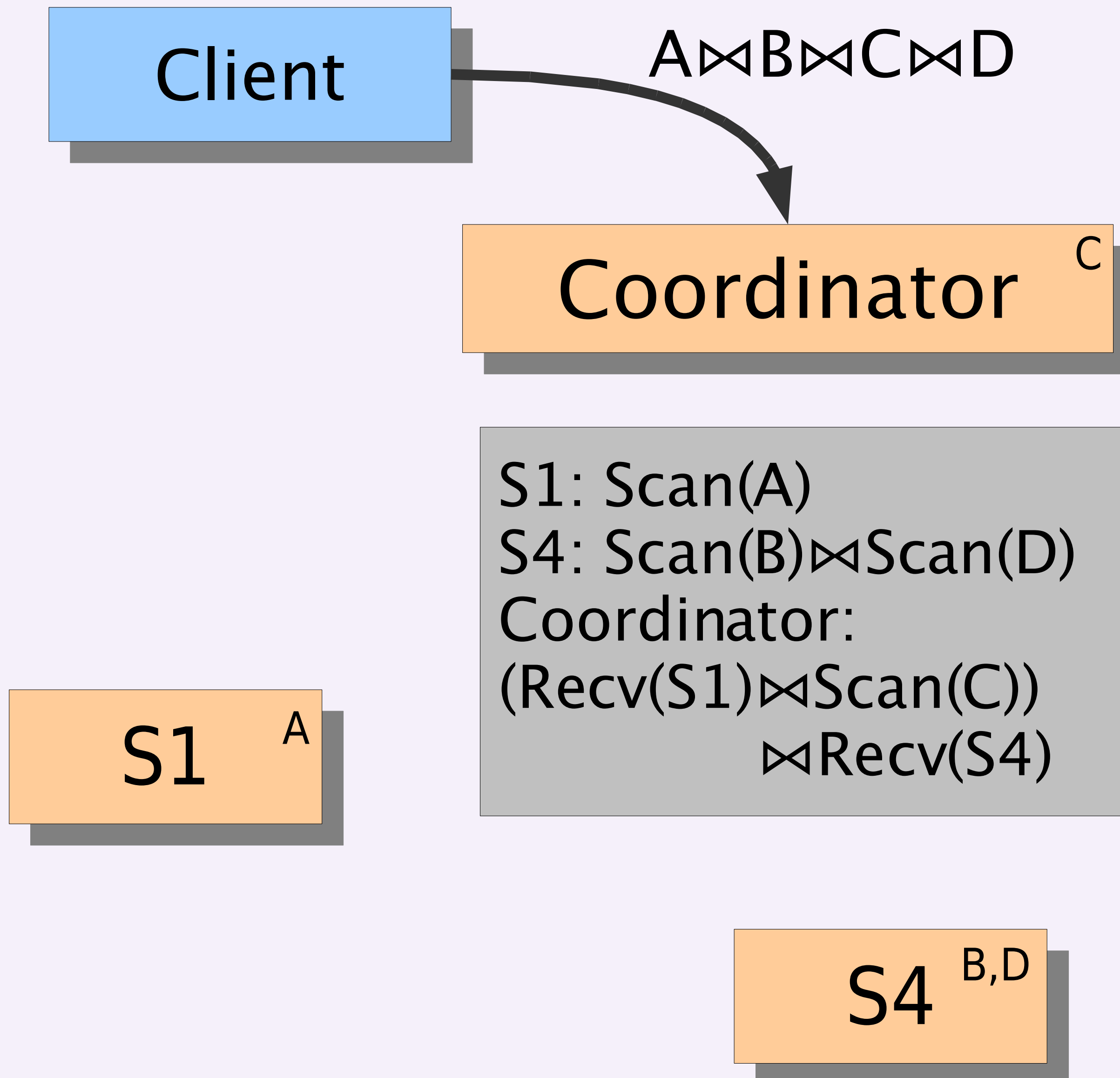
1. Connect to server, submit a query

The textbook approach



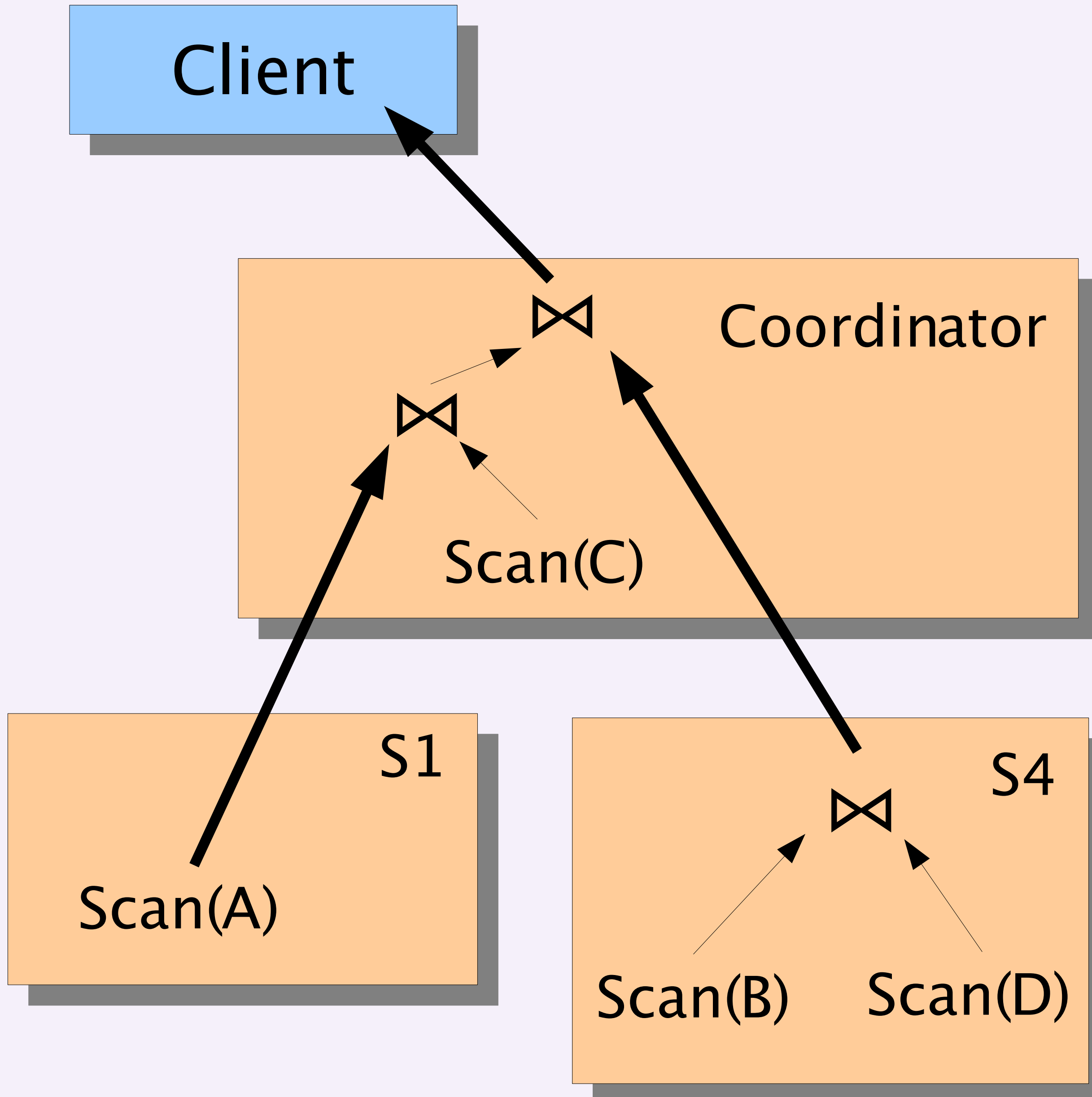
1. Connect to server, submit a query
2. Figure out where relevant data reside

The textbook approach



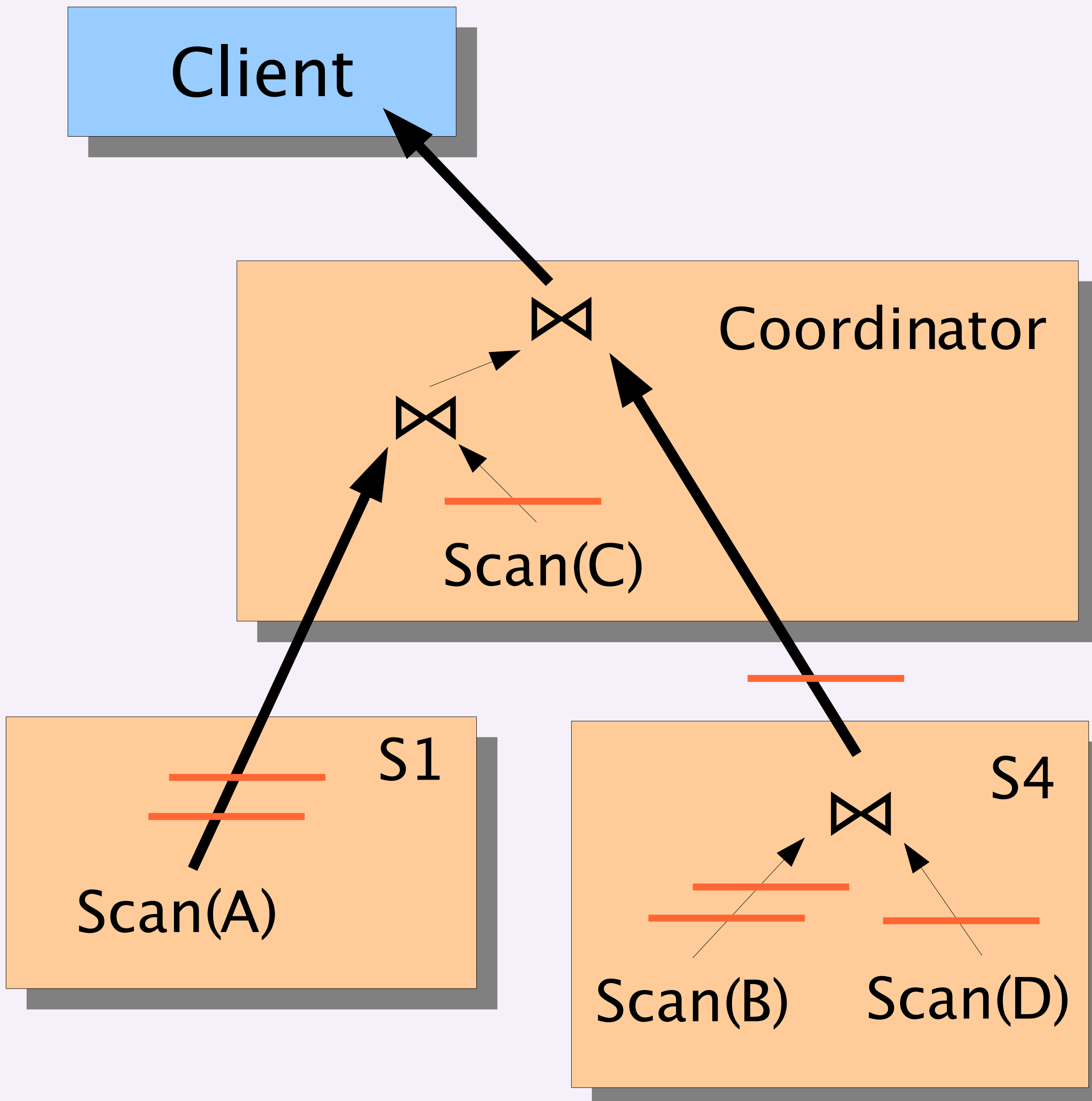
1. Connect to server, submit a query
2. Figure out where relevant data are
3. Optimize query to distributed plan

The textbook approach



1. Connect to server, submit a query
2. Figure out where relevant data are
3. Optimize query to distributed plan
4. Open connections to appropriate servers and deploy sub-plans

The textbook approach



1. Connect to server, submit a query
2. Figure out where relevant data are
3. Optimize query to distributed plan
4. Open connections to appropriate servers and deploy sub-plans
5. Evaluate sub-plans, pipeline tuples through iterators

Many variations & tweaks

- Sub-coordinators
  Coord.: Recv(S1) $\bowtie$ Scan(C)
  S1: Scan(A) $\bowtie$ Recv(S2)
  S2: Scan(B) $\bowtie$ Scan(D)
- Connection graph doesn't have to be a tree
- If some servers take too long to start-up, rearrange the joins to get tuples flowing sooner (query scrambling)
- Non-blocking operator implementations
- Batch tuples when they cross the network

Issues: Cost estimation

- Good cost estimates for simple expressions within a single server
  - Possible to maintain the appropriate statistics, histograms, etc.
- Reduced accuracy for expressions spanning multiple servers
- Reduced accuracy as we increase the number of joins
- May end up choosing a sub-optimal plan

Issues: Catalog accuracy

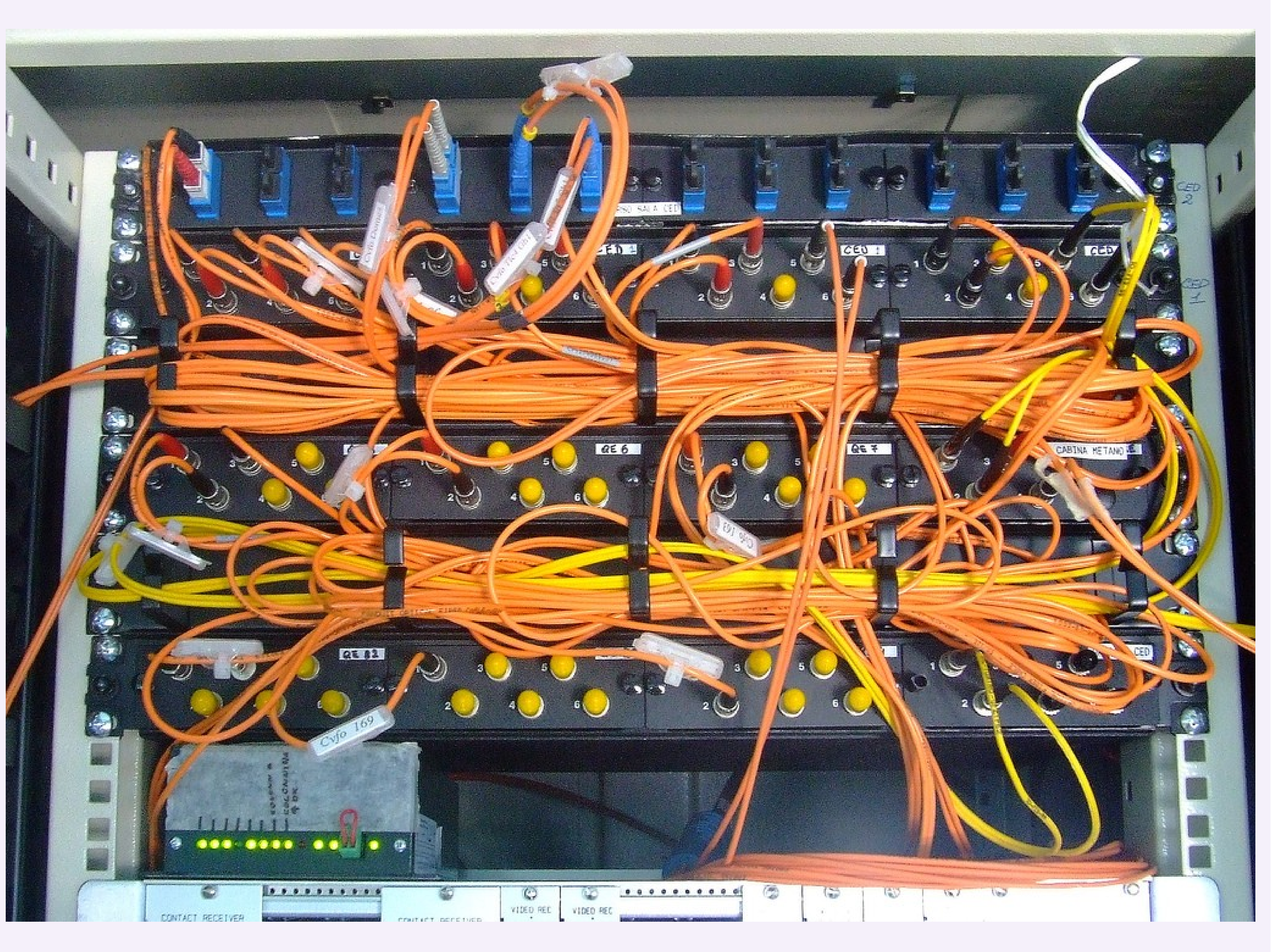
- Hard to keep some catalog info up-to-date
- Replication and fragmentation patterns may change frequently
- Server availability and load may change very frequently
- As conditions change, the optimal plan changes, but optimization decisions are hard to revisit
- A change of plans may force us to throw away some work

Issues: Deployment & Execution

- Deployment can be a *long* process
- Delays on a server affect all servers up the coordination tree, both in deployment and in execution
- A server can't finish until all its inputs are finished
- Fast/underloaded servers must wait for slow/overloaded servers

Issues: Metrics

- Focus on low latency
  - Especially first-tuple latency
- Not always the appropriate metric – what about throughput?
- Resources are being wasted when a server is kept waiting or underutilized
  - Network connections, database connections, allocated buffers aren't free
- Coordination is expensive!



CED 2
CED 1
QE 6
QE 7
CABINA METANO
QE 12
Cvfo 169
CONTACT RECEIVER
VIDEO REC
VIDEO REC

Claim #1

Successful distributed systems implement multi-server interactions as sequences of simple, discrete, point-to-point operations

- TCP connections are made of separate packets, potentially taking different routes
- Web sessions are implemented as a set of HTTP requests, potentially served by different servers or caches

The UI/API can provide the session or connection interface as a layer above a different implementation

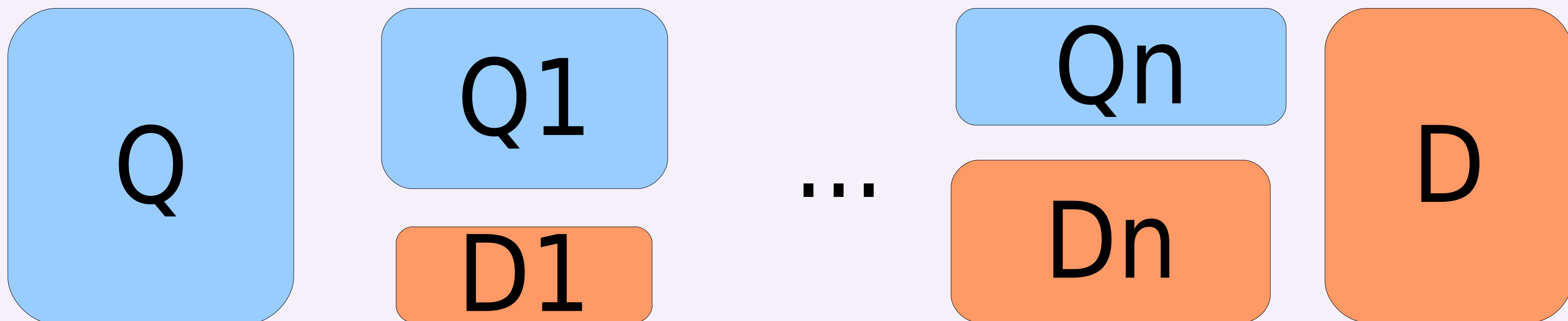
Claim #2

Distributed querying doesn't have to be parallel querying!

- Many convenient assumptions don't hold in a large-scale distributed system (different server capabilities & software, different administrative domains, different goals & priorities)
- Reducing parallelism can reduce the need for coordination and improve throughput

Mutant Query Plans

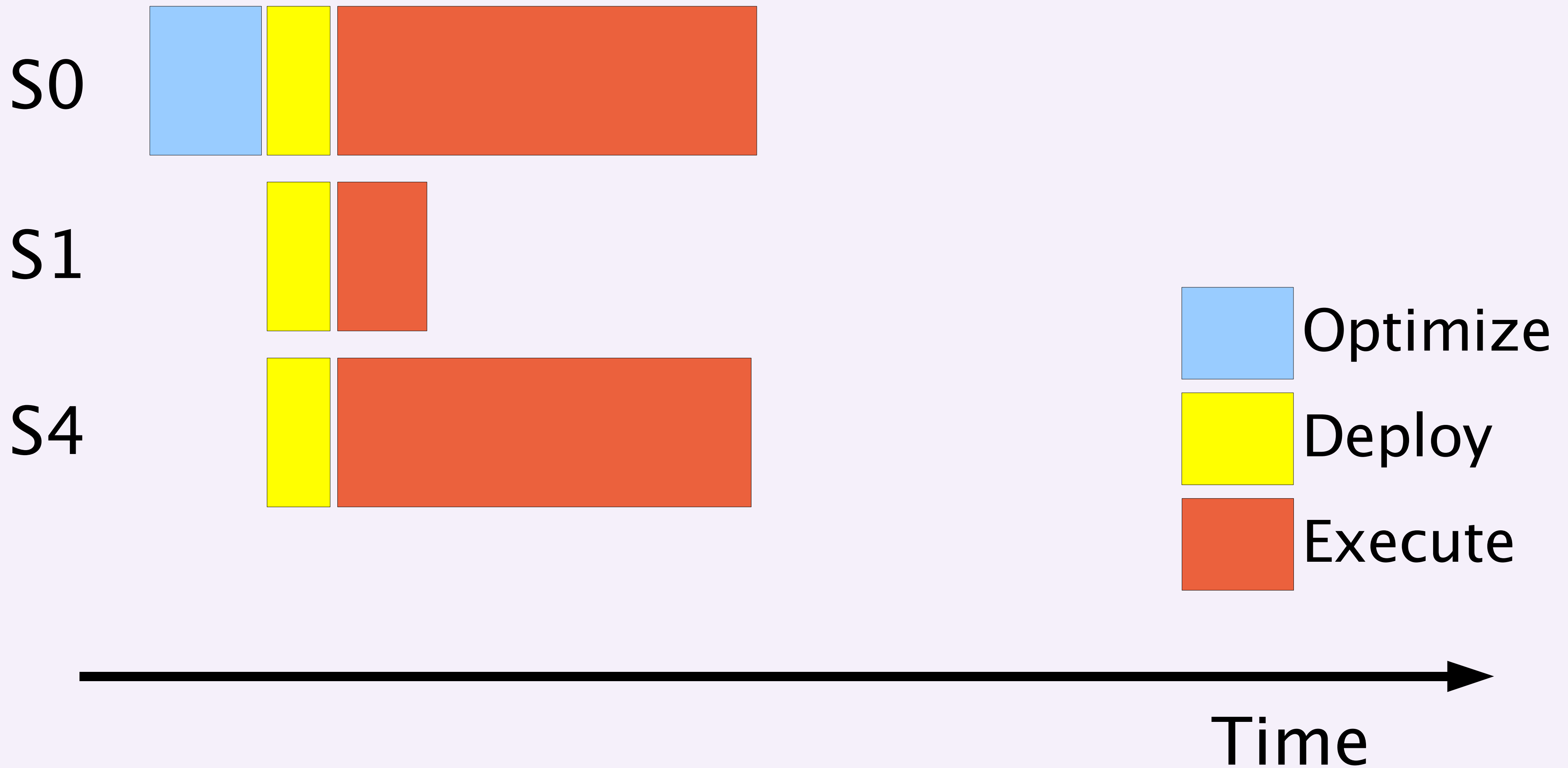
- In the limit, a distributed query plan is active on (at most) one server at a time
- A *Mutant Query Plan* is a bundle of query operators and constant data
- Each server transforms (mutates) the plan by evaluating portions of it, moves the mutated plan to the next server



Evaluation example

Client	$A \bowtie B \bowtie C \bowtie D$	$\Rightarrow$	
S4	$A \bowtie (C \bowtie (B \bowtie D))$		
	$A \bowtie (C \bowtie$ **BD**$)$	$\Rightarrow$	
S0	$A \bowtie$ **CBD**	$\Rightarrow$	
S1	**ACBD**	$\Rightarrow$	Client

Pipelined Plan Timeline



Mutant Plan Timeline

An MQP server prototype

- Built on top of Niagara/NiagaraST
  - XML, Java, in-memory, streaming data, ...
- Shouldn't be hard to build on top of a regular DBMS
- Basic architecture and query engine same as the base Niagara system
- Added Colombia, a Java port of the Columbia query optimizer

Plan encoding

- Plans are encoded in XML, using a paged binary format to reduce parsing overhead (based on Niagara's in–memory representation)
- Leaves of the operator graph can be:
  - regular XML data
  - URLs
  - URNs
- In a relational setting
  plan could be SQL query + (temp. tables)

Resource Resolution

- Catalog maps each resource/URN to:
  - A set of URLs (local files, HTTP URLs, URLs served by MQP servers). Each URL represents a horizontal fragment of the resource.
  - the addresses of servers that may be able to resolve the URN further
- Catalog also knows about fragment replication (equivalences between URLs)
- There are queries where full resolution is not always possible (or desirable!)

Optimization

- Logical expressions can be *intensional* (e.g., unresolved resources) or *extensional*
- An intensional expression cannot be fully evaluated locally (although we may be able to estimate its cardinality & size)
- Optimizer decides which extensional expressions to evaluate locally and where to route the plan to next
- Cost metric is (estimated) global CPU + network cost

Optimization (2)

- Have accurate statistics for expressions that have been evaluated in previous servers
- Further assumptions:
  - can accurately estimate the cardinality and size of locally evaluated expressions
  - can make educated guesses about expressions involving unresolved resources (as good as the coordinated optimizer)

Consolidation

- Transformation rules move extensional expressions together, allowing more work to be done locally

  $((\mathbf{A} \bowtie X) \bowtie (\mathbf{B} \bowtie Y)) \bowtie \mathbf{C} \quad \Rightarrow$

  $((\mathbf{A} \bowtie \mathbf{B}) \bowtie \mathbf{C}) \bowtie (X \bowtie Y)$

- Full consolidation not always possible
- Sometimes undesirable (e.g., if it introduces cartesian products)

Absorption

- Depending on output sizes, we may want to combine intensional & extensional portions of a plan even if we can't achieve consolidation
- Example rule:
  if $|A \bowtie B| < |A|$
  $(\mathbf{A} \cup X) \bowtie \mathbf{B} \Rightarrow (\mathbf{A} \bowtie \mathbf{B}) \cup (X \bowtie \mathbf{B})$

Deferment

- Sometimes it is preferable to defer the evaluation of an extensional expression for another server (e.g., when a resource fragment is available both locally and on the next server)
- *DEFER(x)* materializes the results of x's inputs and defers the evaluation of x

Routing choice

- Options for next server: servers that can resolve URNs, servers that have relevant URLs
- Compute a separate cost per physical operator for each such routing choice
  - penalize shipping data to servers that already have them or can compute them
- Choose the next server associated with the cheapest plan for the complete query expression

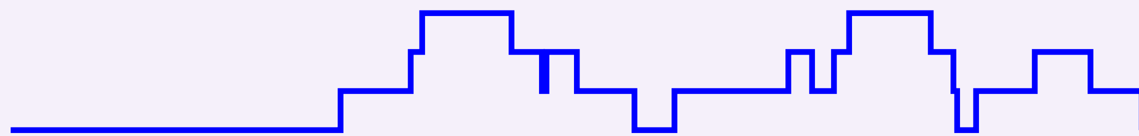
Evaluation & Routing

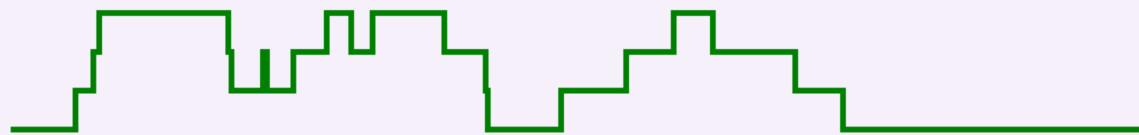
- Evaluate all non-deferred sub-plans from the chosen plan and materialize the results as separate XML documents
- Replace each evaluated sub-plan with a its results (or by a URL to them!)
- Possible to revisit routing choice at this point by repeating optimization
- Ship to the next server using HTTP, or send to the client if evaluation is complete

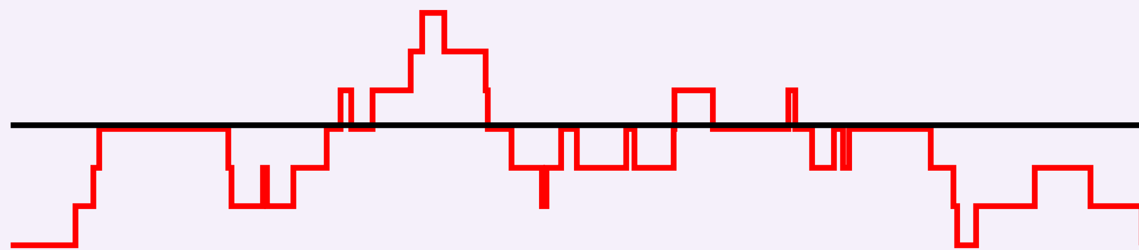
Pipelined and Mutant Plans in Undependable Environments

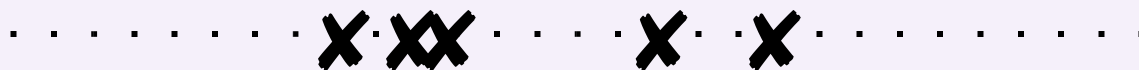
- Loose federation of servers, no central authority
- High-priority local workload, low-priority remote queries
- A server may be *unavailable* for some period of time (new queries are not admitted, current queries can continue)
- A server may occasionally *terminate* a query, losing related state and forcing a restart

Availability and Terminations





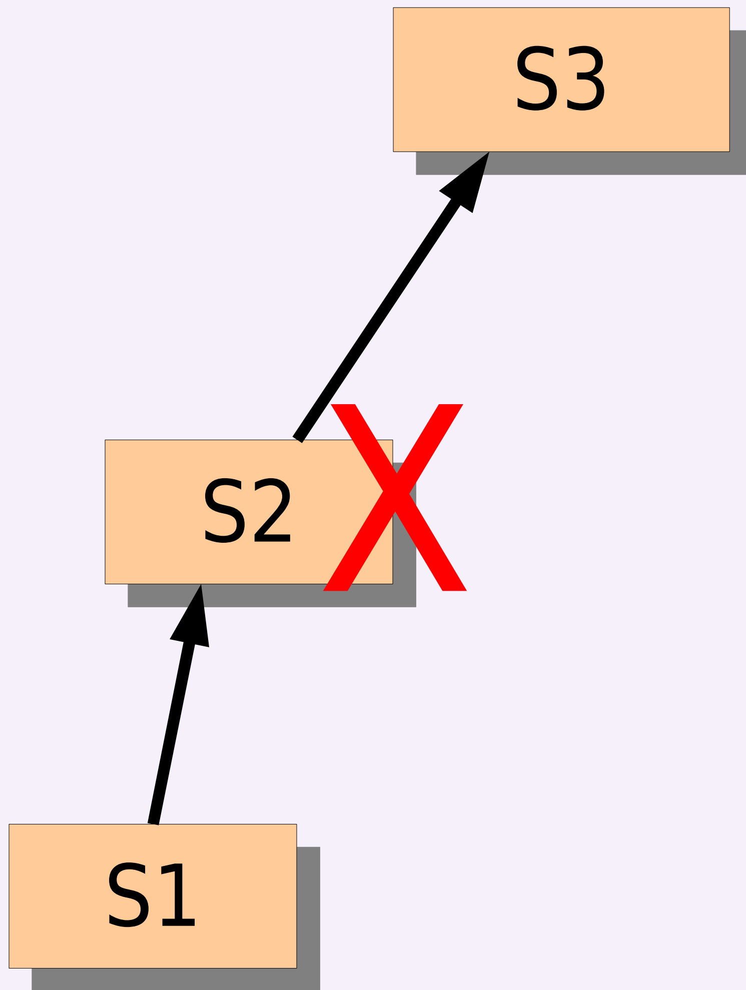




Distributed plan flavors

- Pipelined plans (PP) and mutant plans (MQP) do business as usual, but must restart from scratch when a termination occurs on a server they're running on
- A restartable pipelined plan (RPP) buffers remote inputs, but needs to restart just the sub-plan rooted on the terminated server
- A checkpointed mutant plan (CMQP) restarts the plan from a backup copy kept on the previous server on the plan's route

Termination behavior



PP: Restart on all three servers

RPP: Restart on S2 and S1

MQP: Go back to S1

CMQP: Use the checkpoint on S1 to restart on S2

Simulation parameters

- Simplification – consider the performance of a *single* query, simulate availability and terminations with two independent random processes
- Intervals for changing server availability follow geometric distributions, with means ***a*** and ***u*** (measured in # of simulation steps)
- Interval between consecutive terminations also geometric, with mean λ (also measured in # of simulation steps)

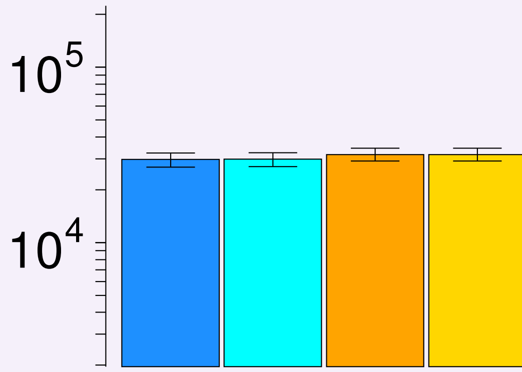
Metrics

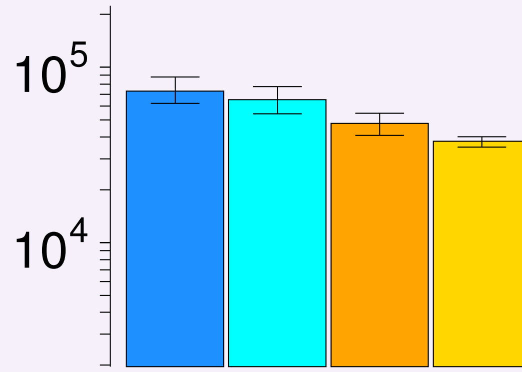
- Elapsed time (# of simulation steps)
- Residency (total steps across all servers)
- CPU usage (total # of CPU instructions)
- Storage footprint (integral over time of the number of tuples buffered on each server)
- Network usage (total # of tuples transferred)

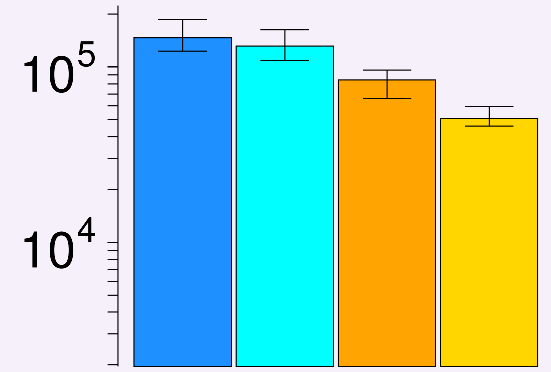
Varying availability ratio (a:u) and termination frequency λ

- Left-deep five-way join
  $(((A \bowtie B) \bowtie C) \bowtie D) \bowtie E$
- $|A|$ = 1 million tuples
- $|B| = |C| = |D| = |E|$ = 10 million tuples
- Each relation on a separate server
- Keep the length of the availability cycle (a+u) constant, vary a:u and λ

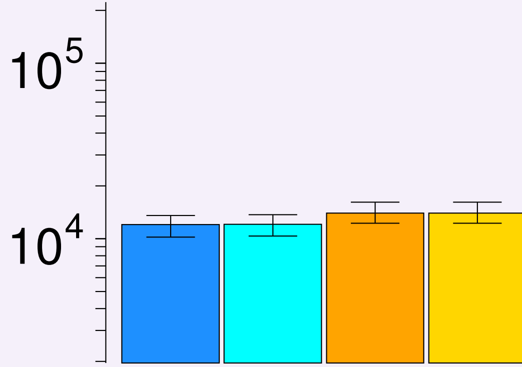
a=2000
u=8000

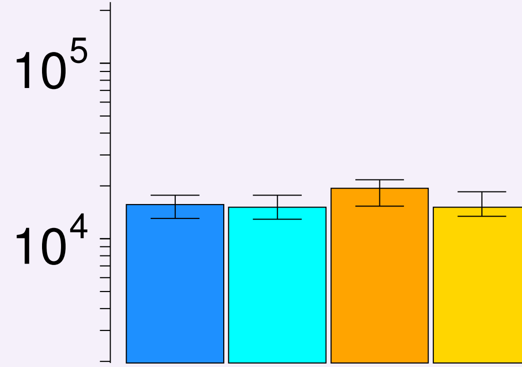


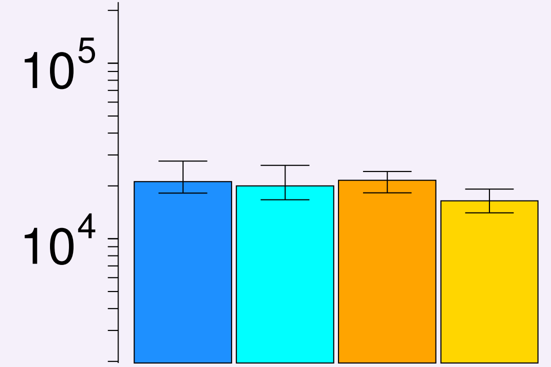




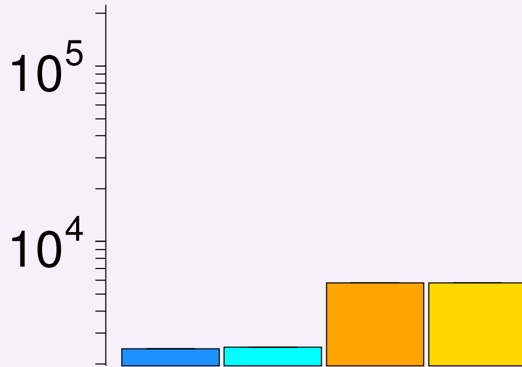
a=5000
u=5000

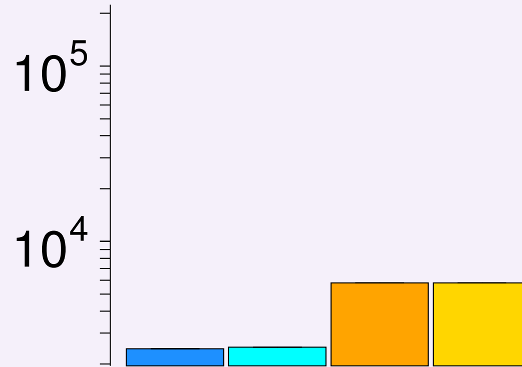


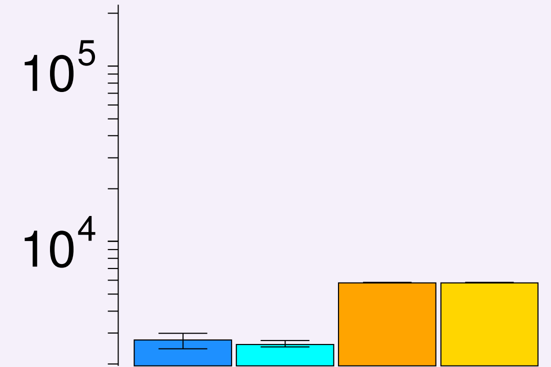




$a = \infty$

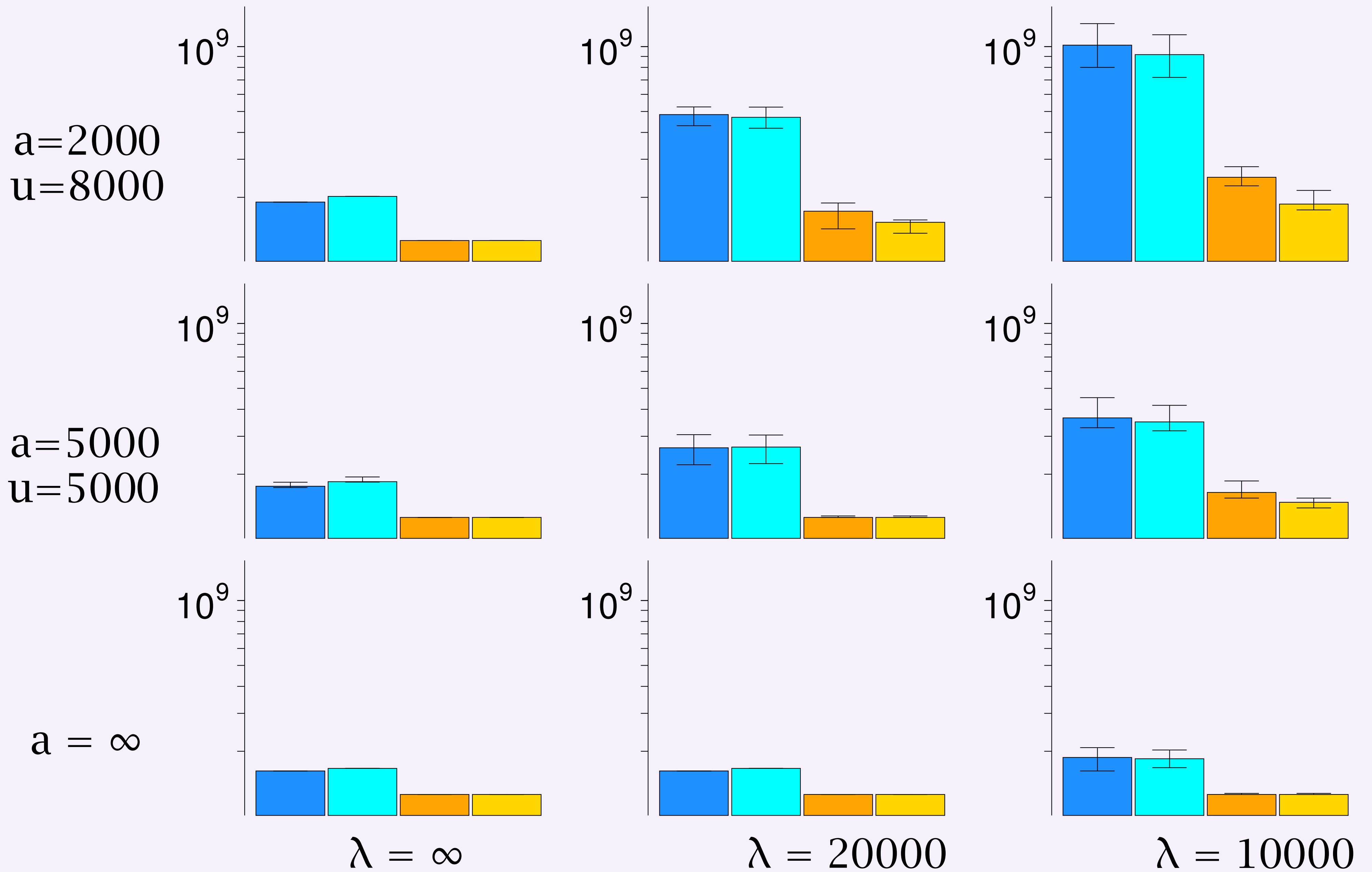






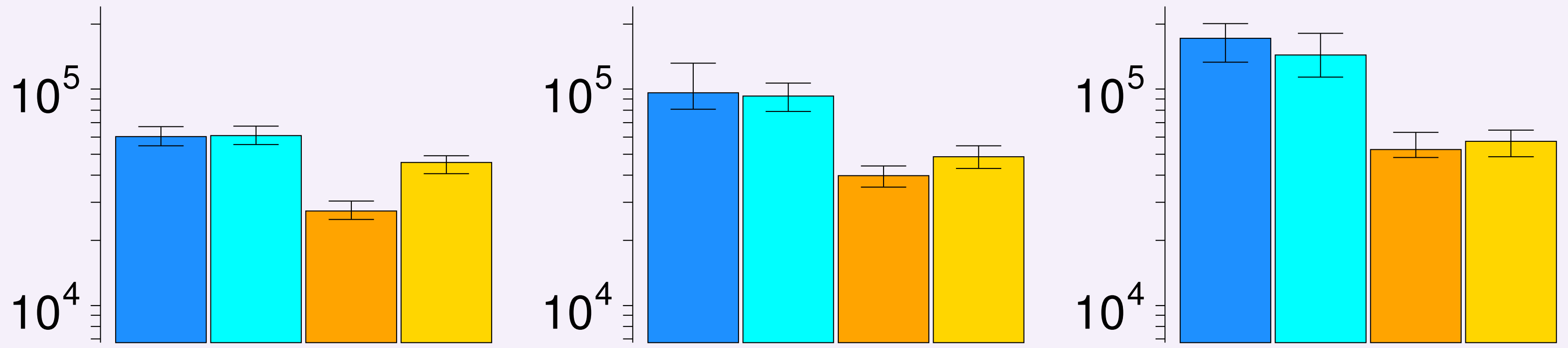
$\lambda = \infty$ $\lambda = 20000$ $\lambda = 10000$

Elapsed time of a 5-server left-deep join
median of 200 runs (in simulation steps)

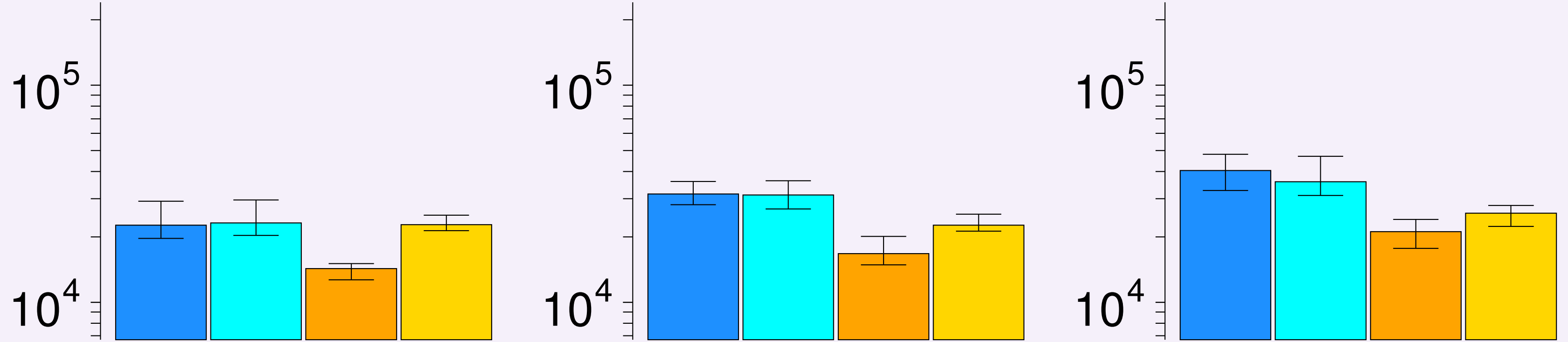


CPU usage of a 5-server left-deep join
median of 200 runs (in instructions)

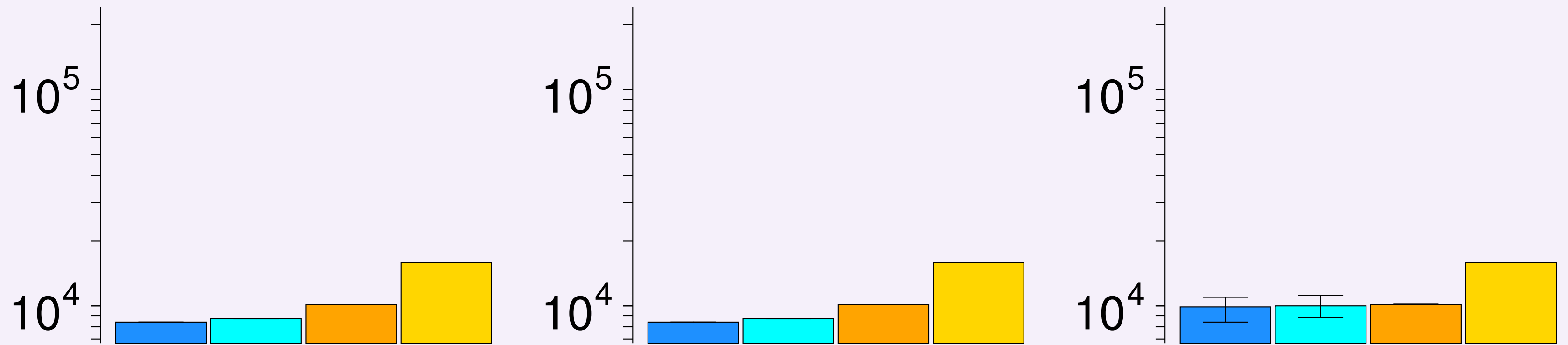
a=2000
u=8000



a=5000
u=5000



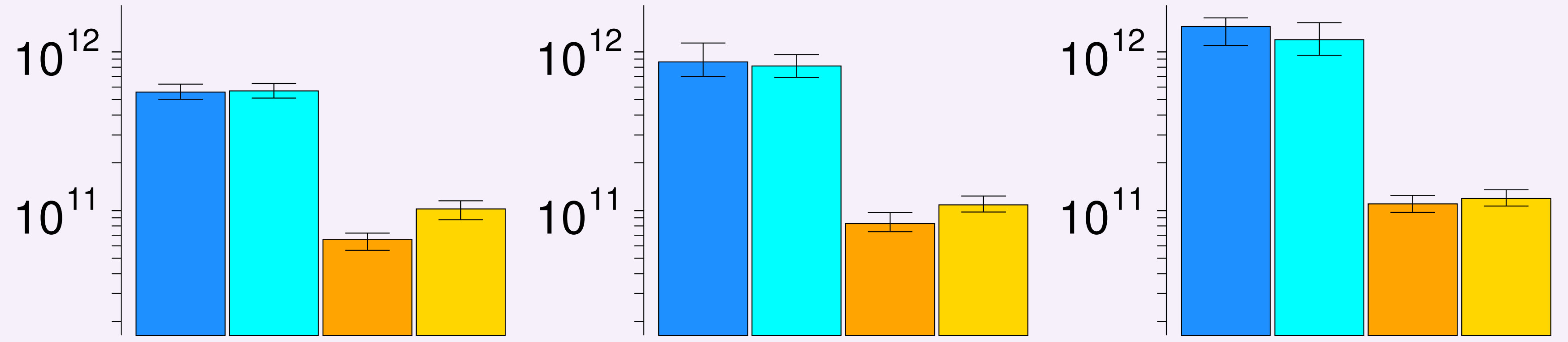
$a = \infty$



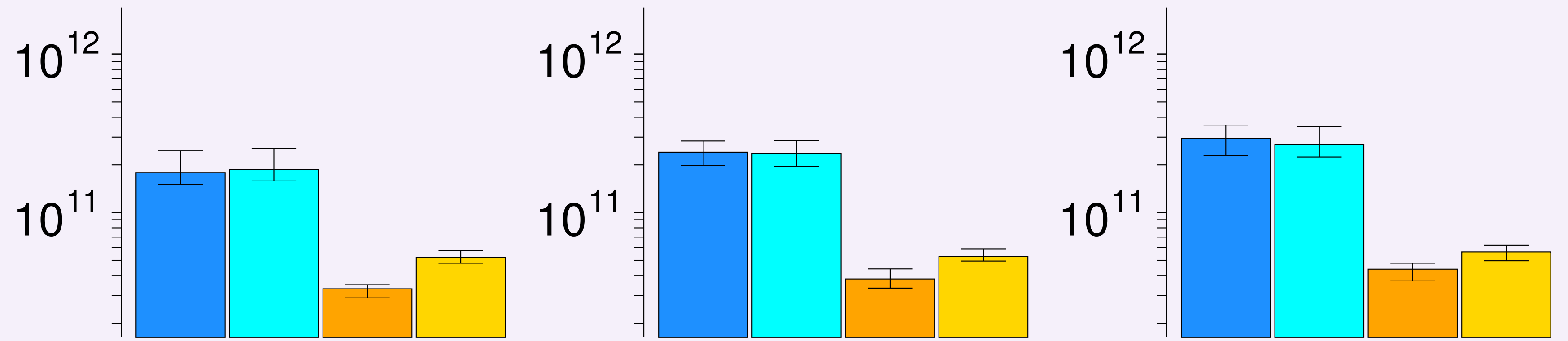
$\lambda = \infty$ $\lambda = 20000$ $\lambda = 10000$

Residency of a 5-server left-deep join
median of 200 runs (in simulation steps)

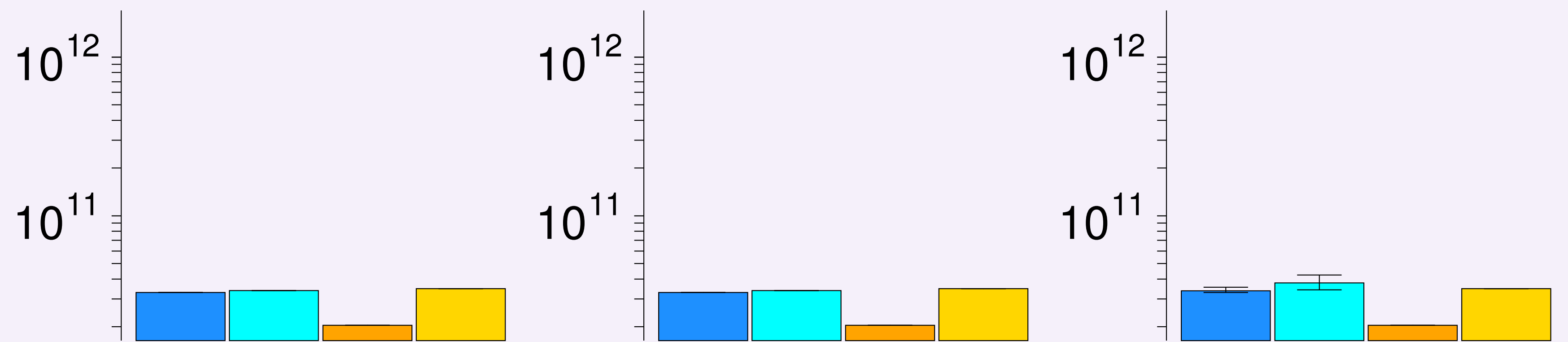
a=2000
u=8000



a=5000
u=5000



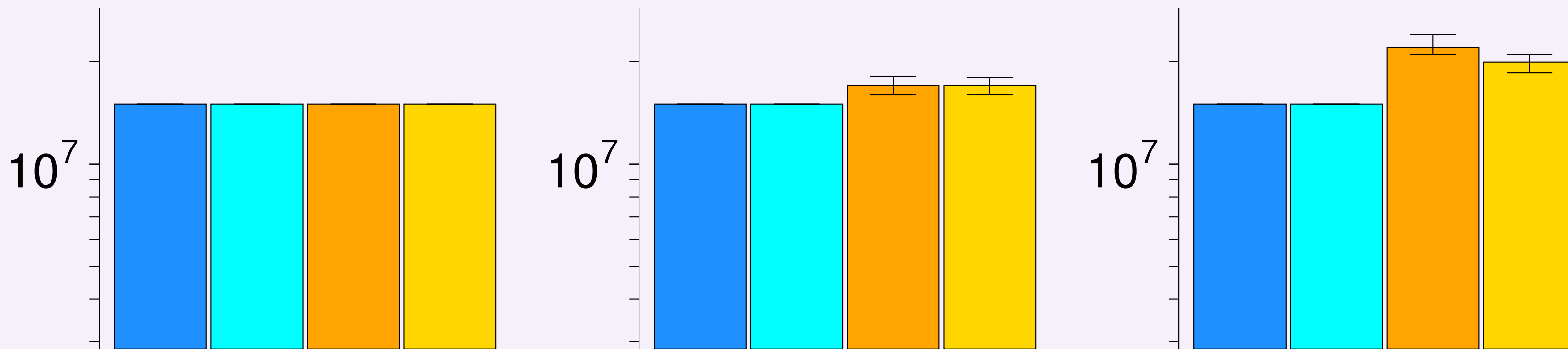
$a = \infty$



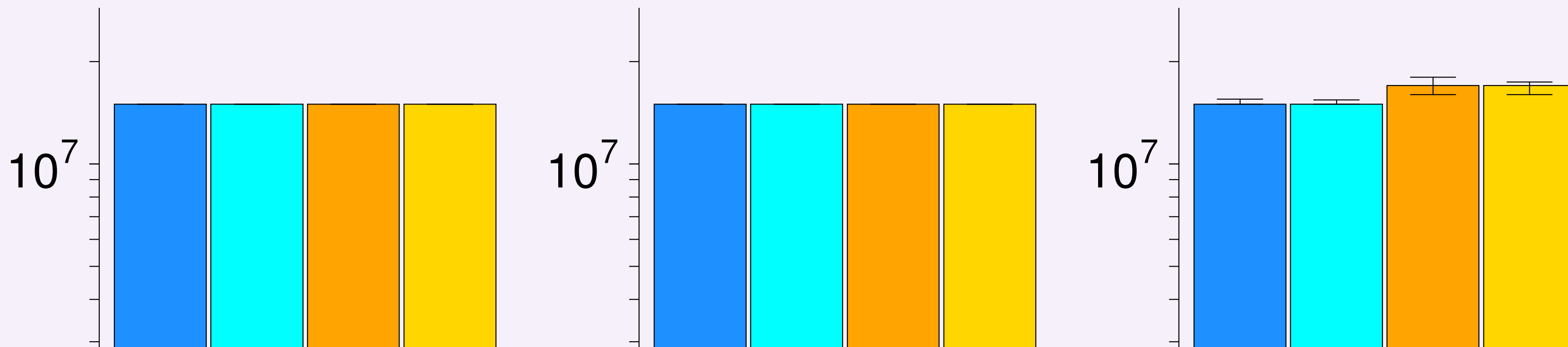
$\lambda = \infty$ $\lambda = 20000$ $\lambda = 10000$

Footprint of a 5-server left-deep join
median of 200 runs (in tuples*simulation steps)

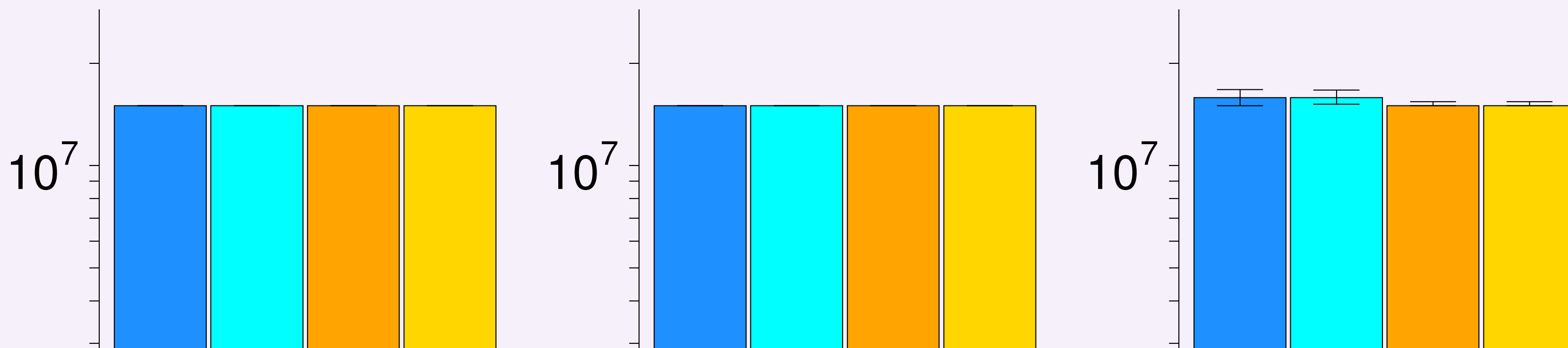
a=2000
u=8000



a=5000
u=5000

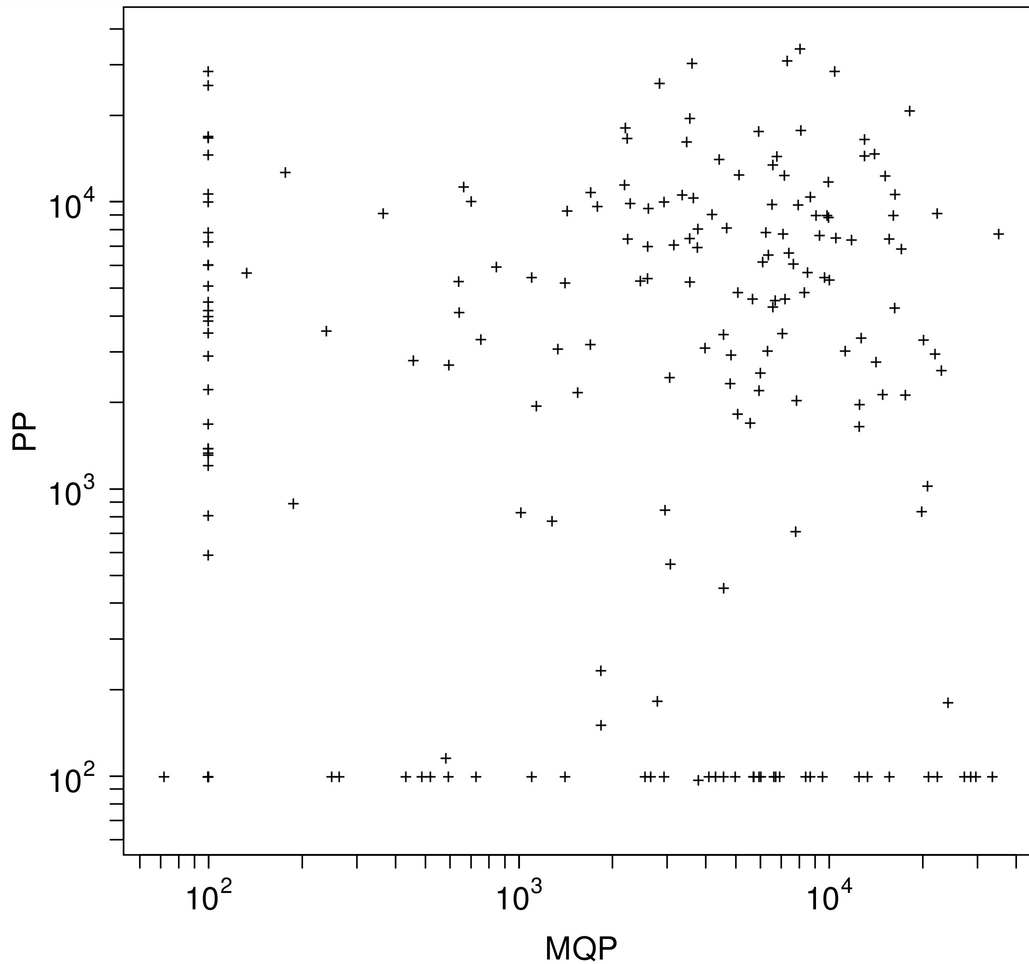


$a = \infty$

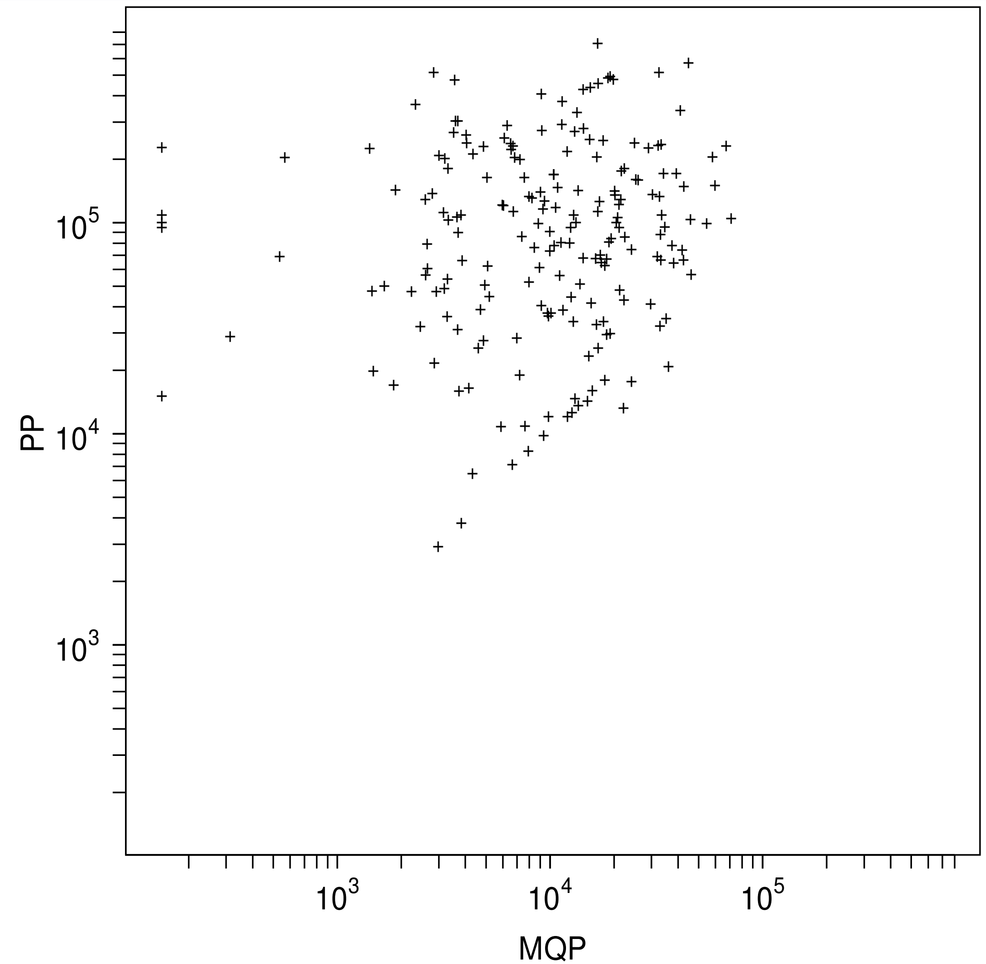


$\lambda = \infty$ $\lambda = 20000$ $\lambda = 10000$

Network usage of a 5-server left-deep join
median of 200 runs (in tuples)



First tuple processed



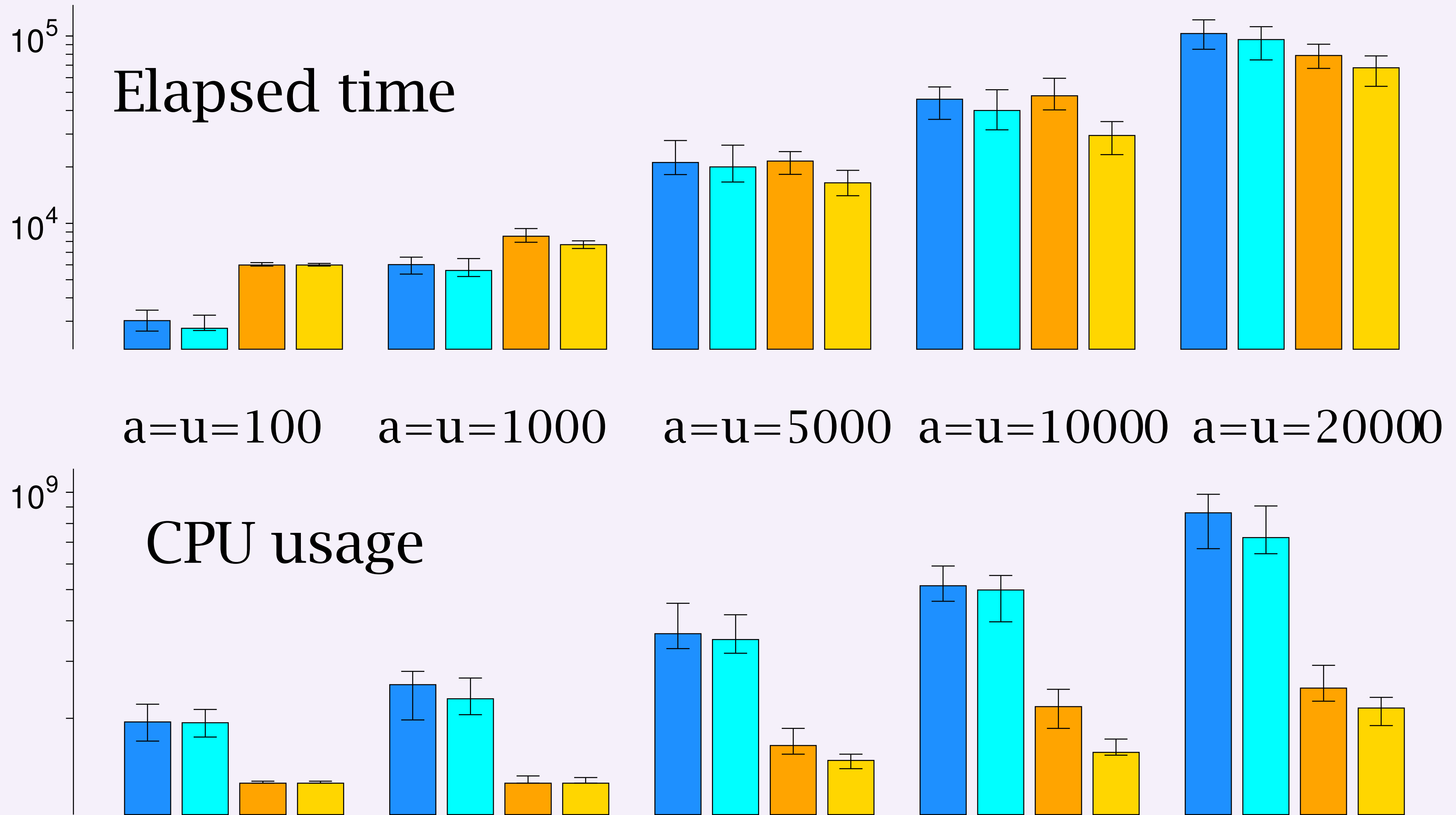
First tuple transmitted

MQPs have increased network usage in low-availability and frequent-termination scenarios because on average they begin transmitting tuples sooner

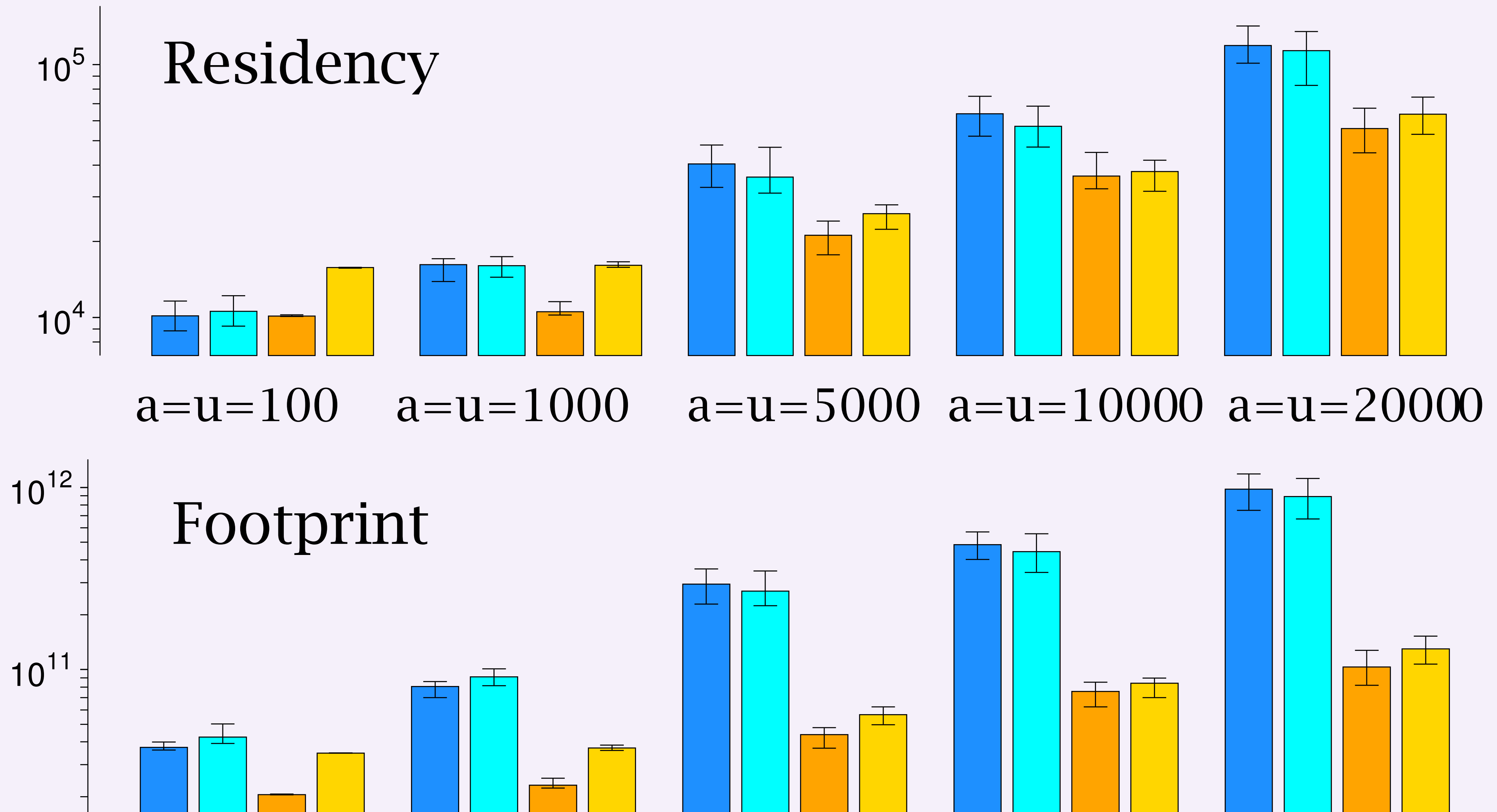
Availability cycle duration (a+u)

- Same query: $(((A \bowtie B) \bowtie C) \bowtie D) \bowtie E$
- $|A|$ = 1 million tuples
- $|B| = |C| = |D| = |E|$ = 10 million tuples
- Each relation on a separate server
- Keep $a=u$ and $\lambda=10000$ steps, vary $a+u$

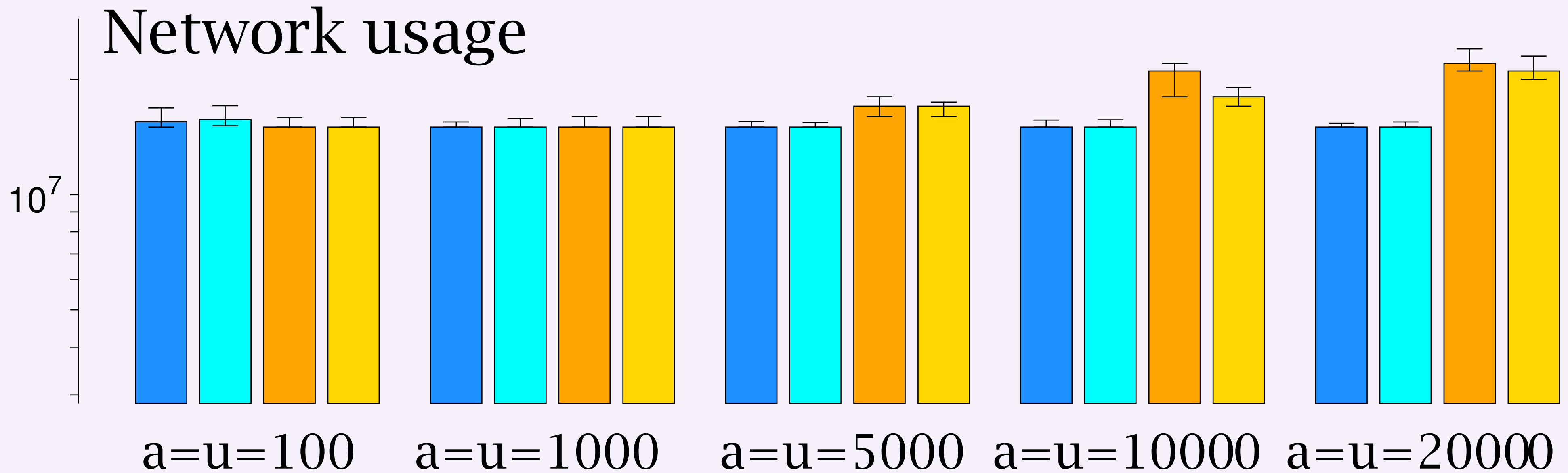
Availability cycle duration (a+u)



Availability cycle duration (a+u)



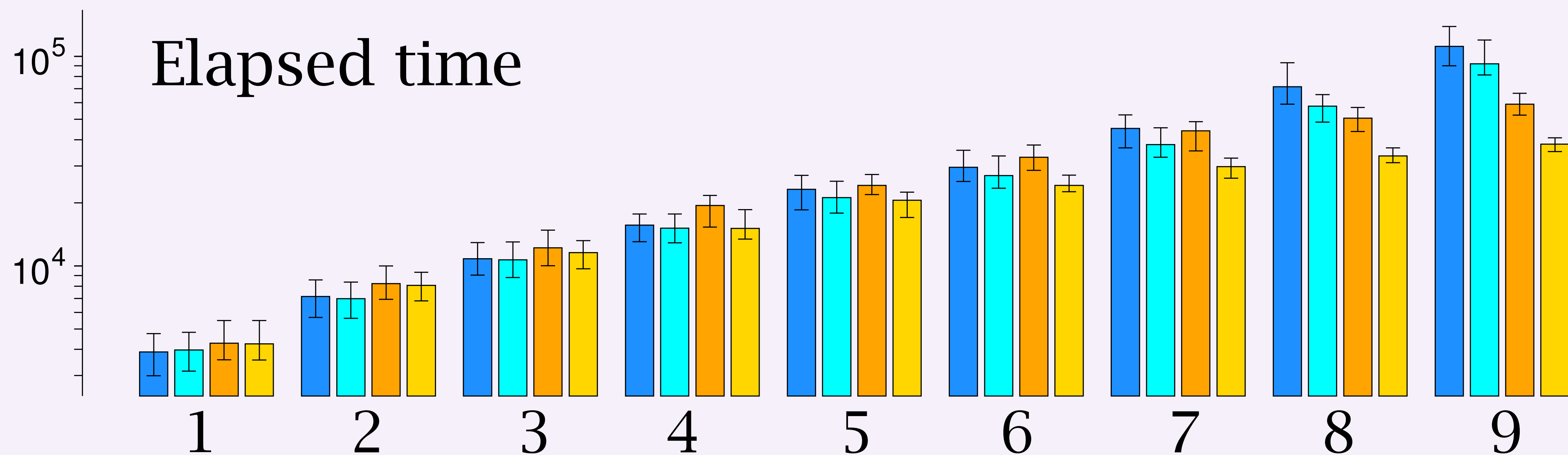
Availability cycle duration (a+u)

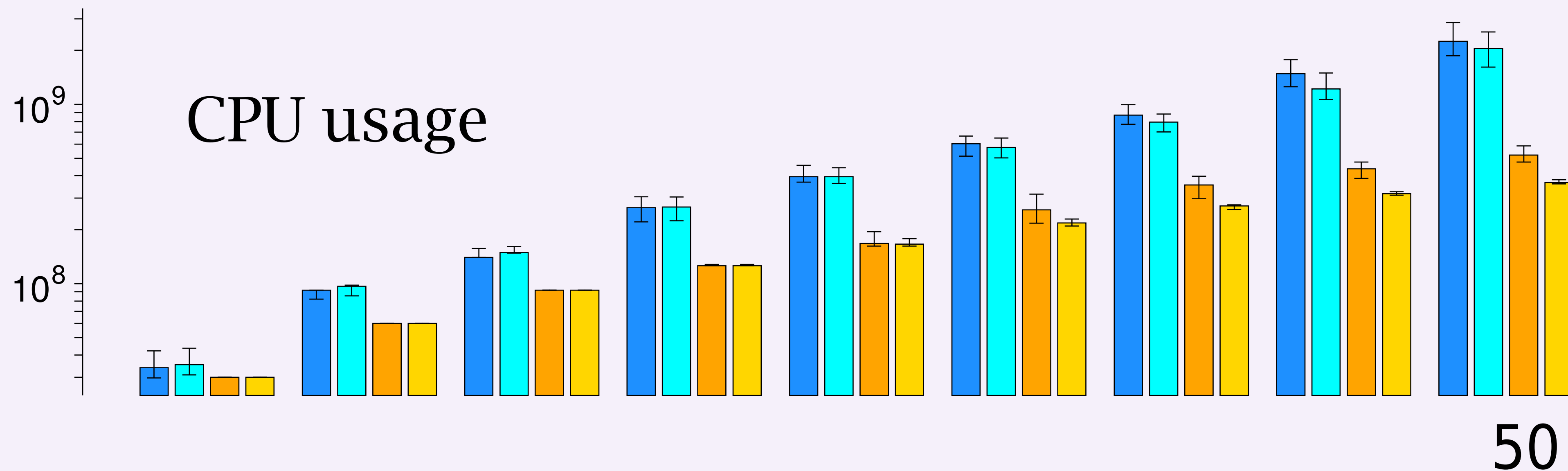


Join depth

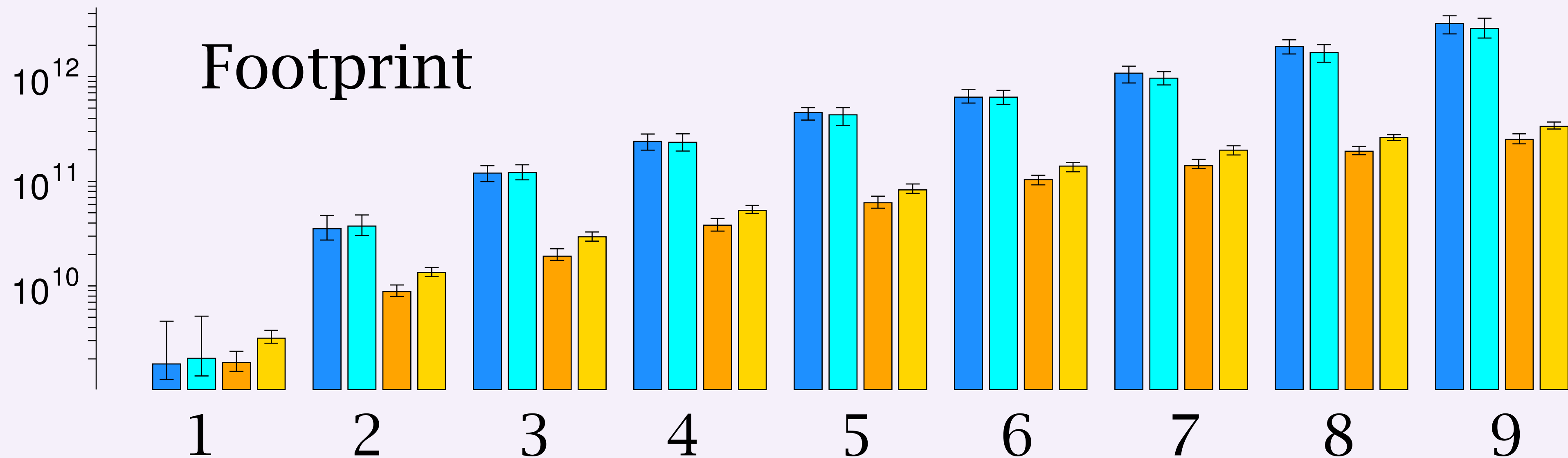
- Keep a=u=5000 steps, λ=20000 steps
- Vary number of joins between 1 and 9
- Still left-deep (((A⋈ B)⋈ C)⋈...)
- A is 1 million tuples, all other relations 10 million tuples

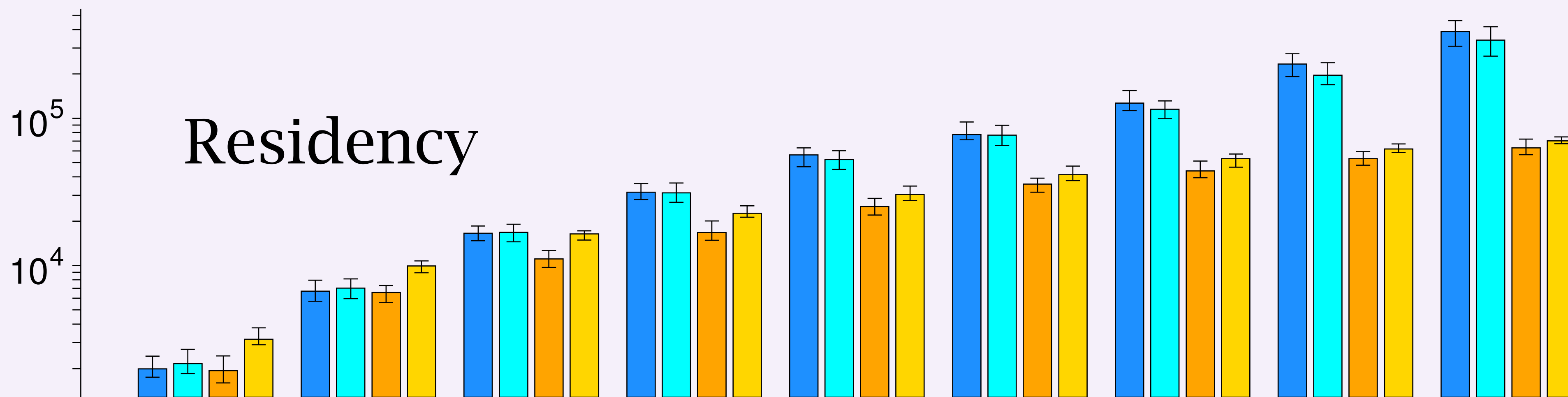
Join Depth



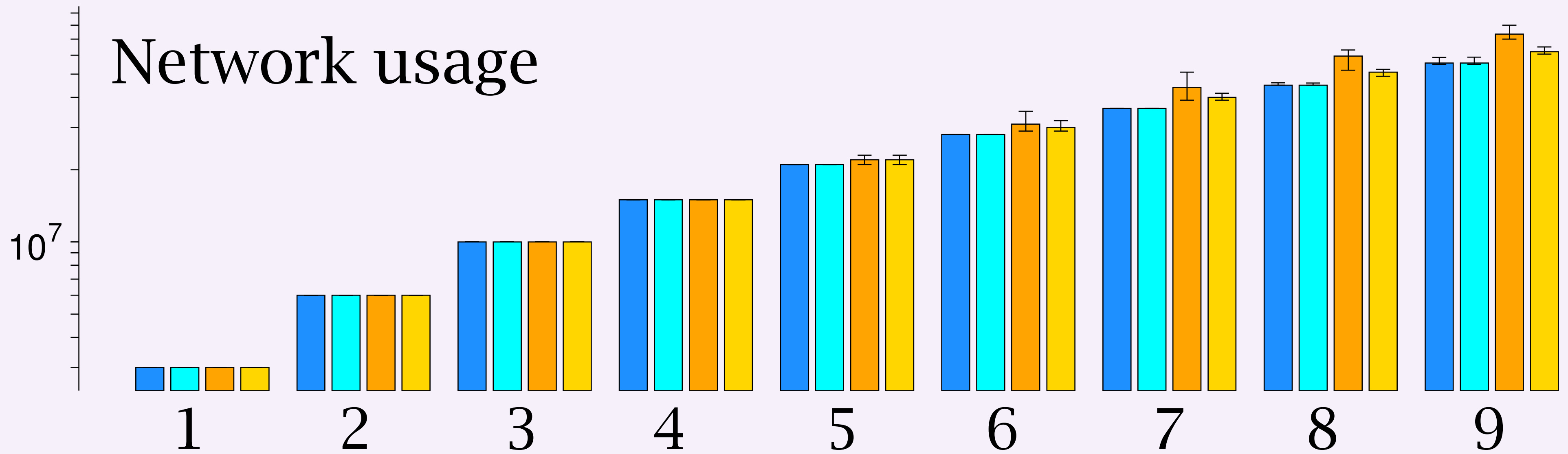


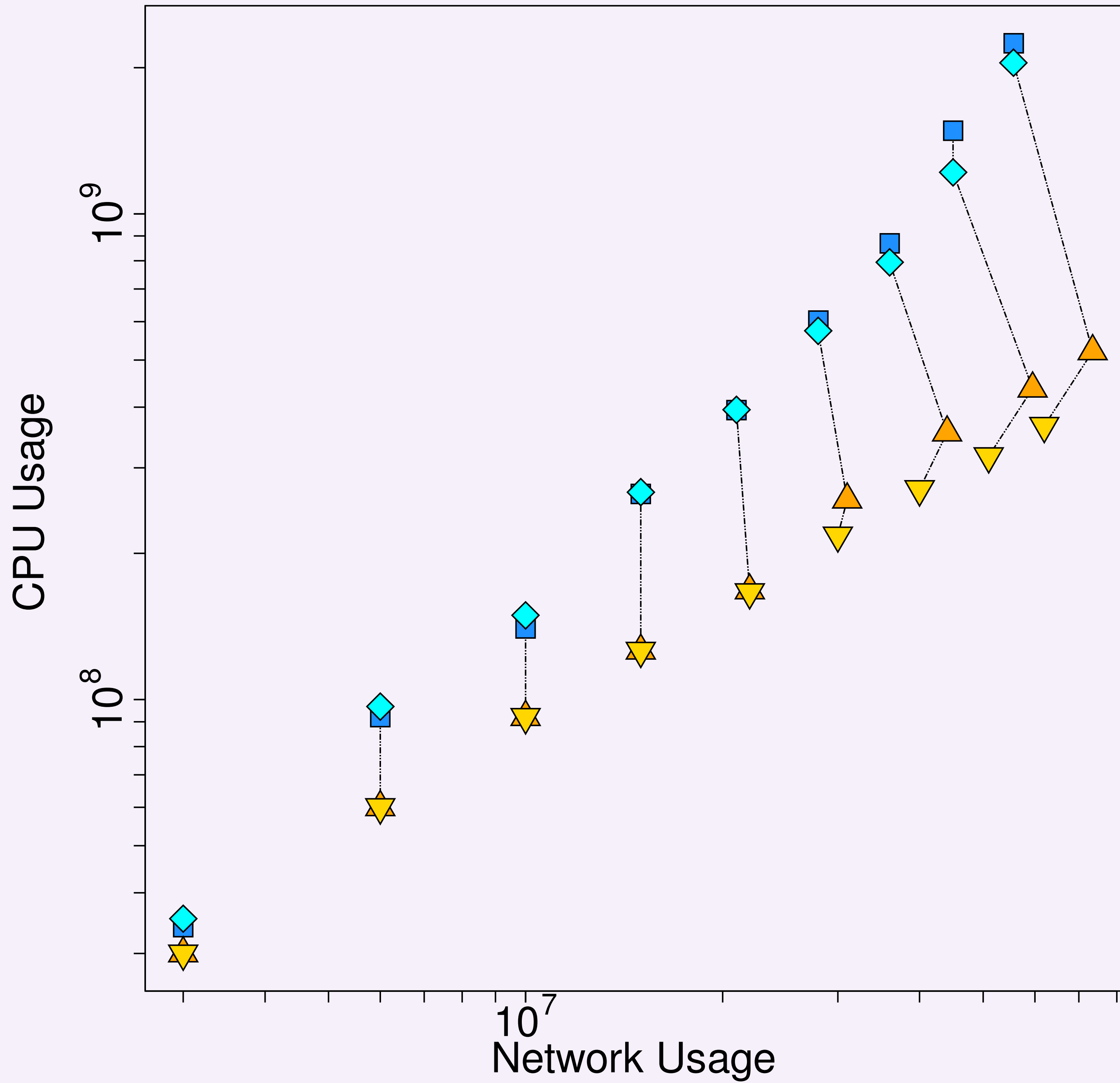
Join Depth

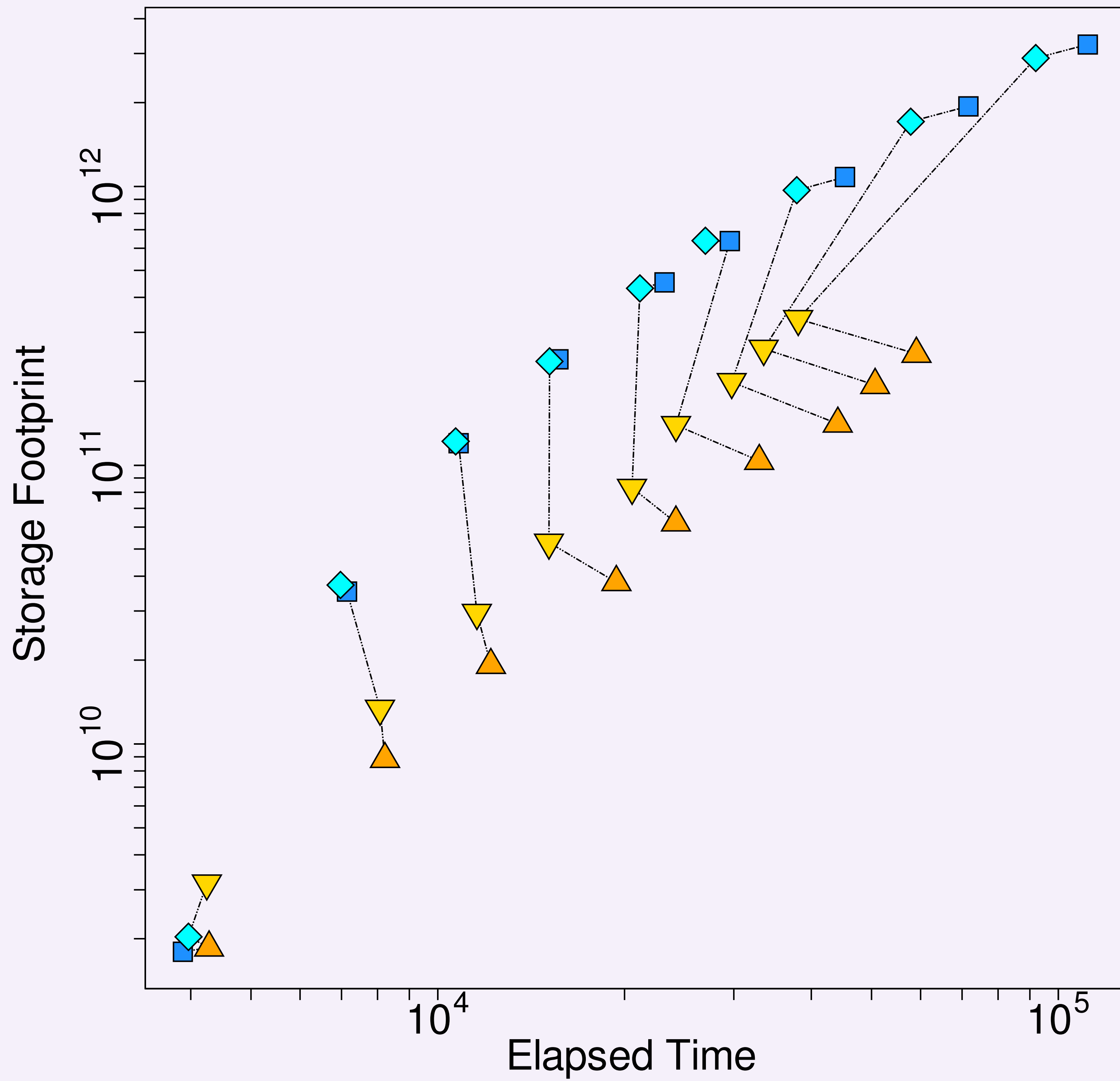




Join Depth



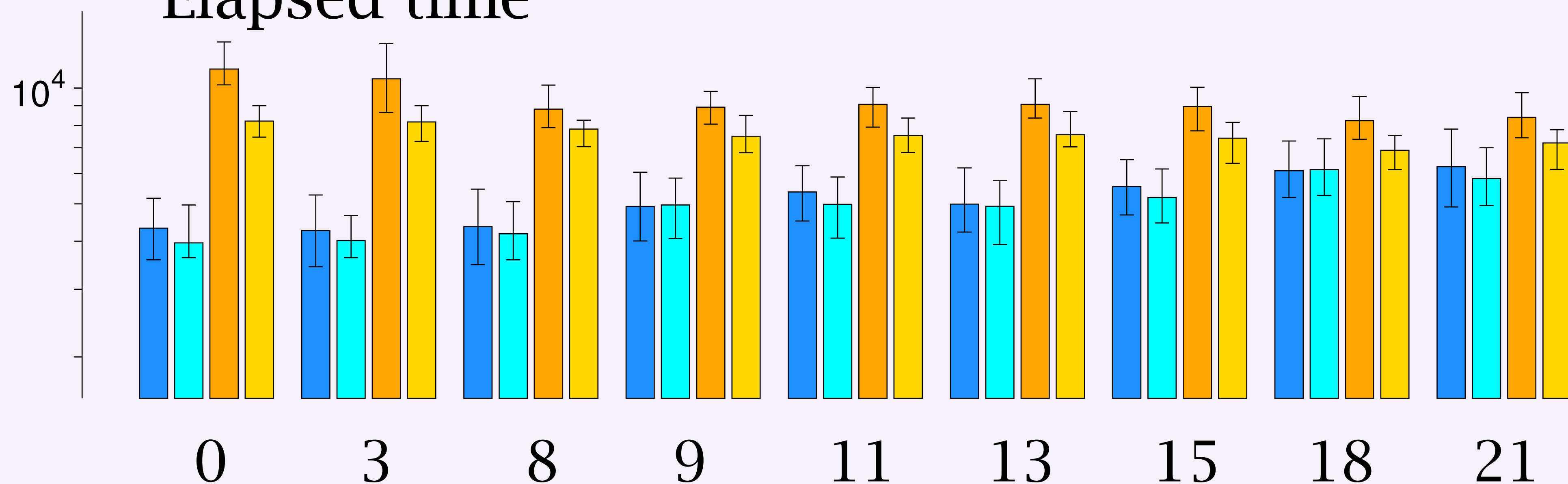

CPU Usage
10^9
10^8
10^7
Network Usage

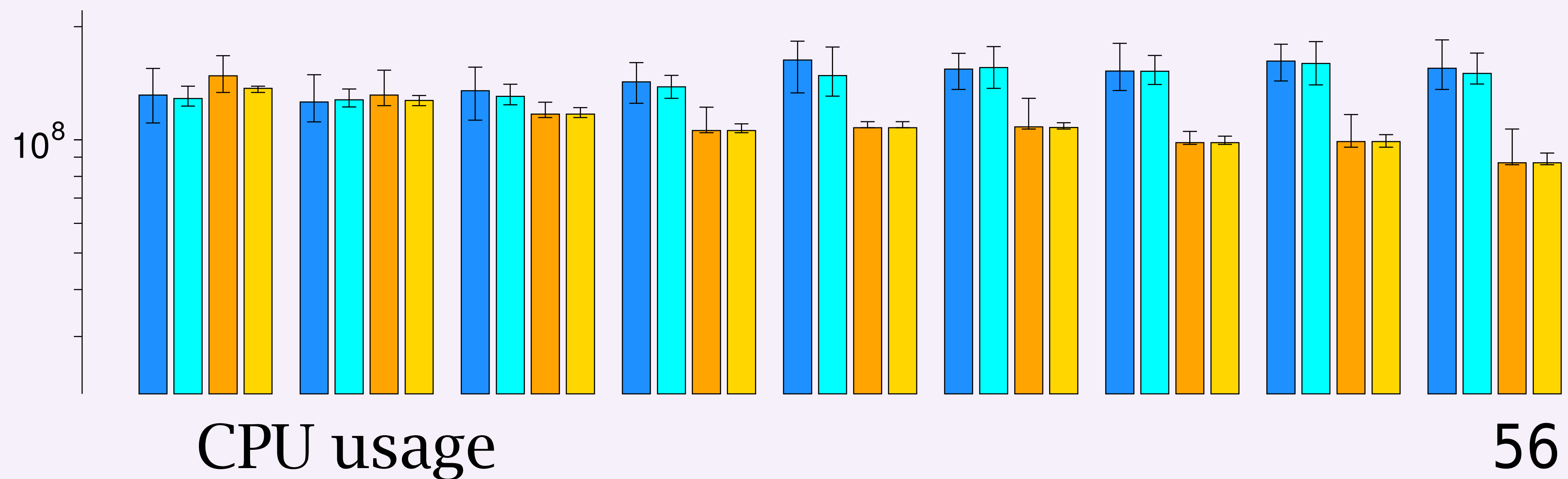

Storage Footprint
10^{10}
10^{11}
10^{12}
10^4
10^5
Elapsed Time

Tree (im)balance

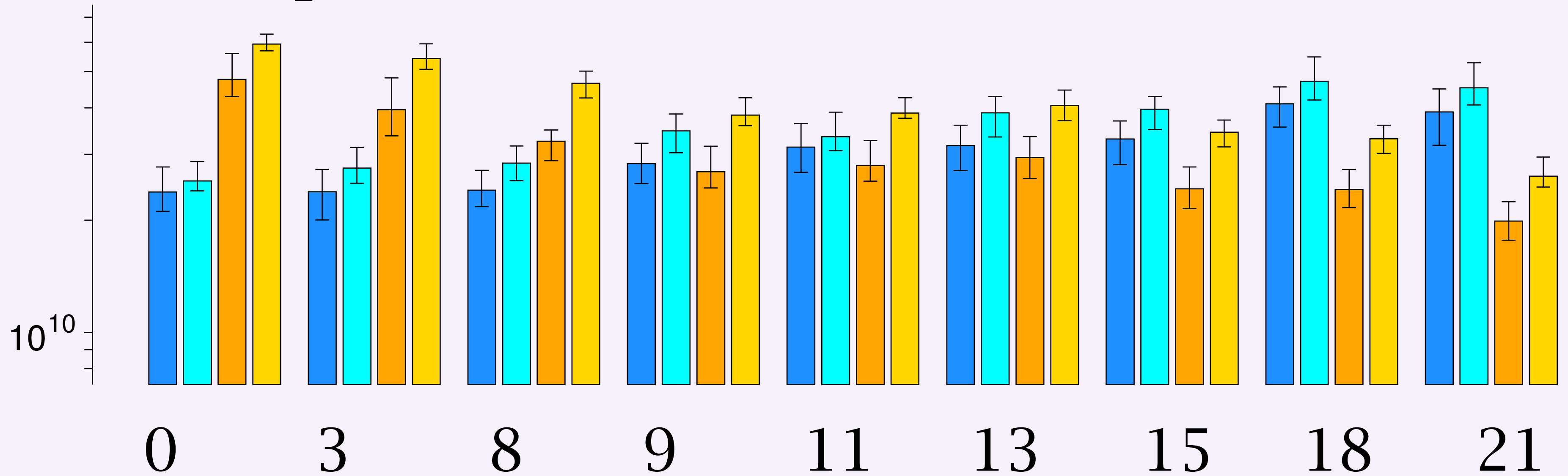
- a=7500, u=5000 , λ=10000 steps
- 8-way joins of varying balance
- Fix selectivities so that all pipelined plans transfer the same # of tuples (mutant plans may transfer more for bushy joins)
- For each balance level, pick the join with the smallest median elapsed time for RPP

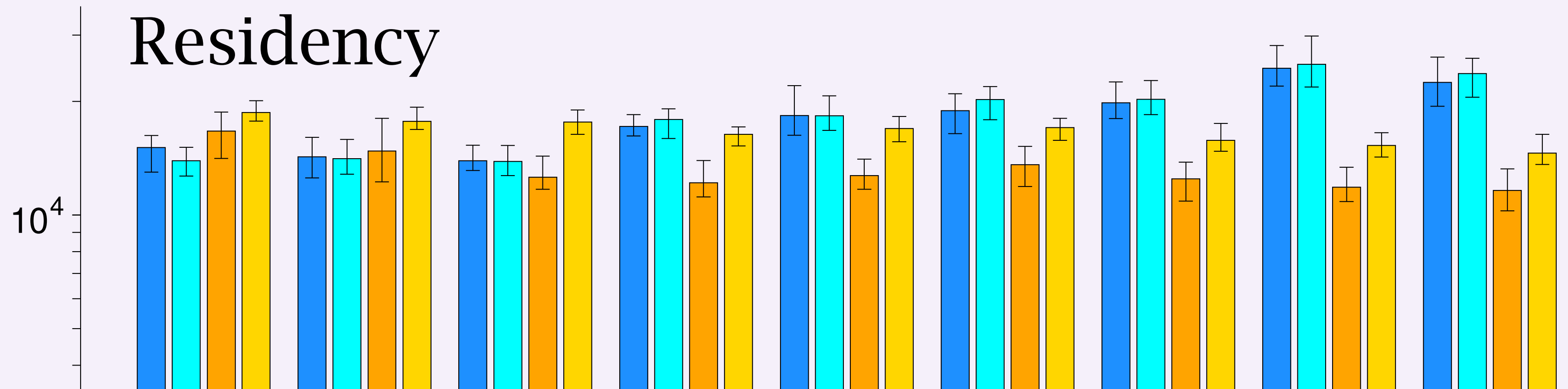
Tree (im)balance



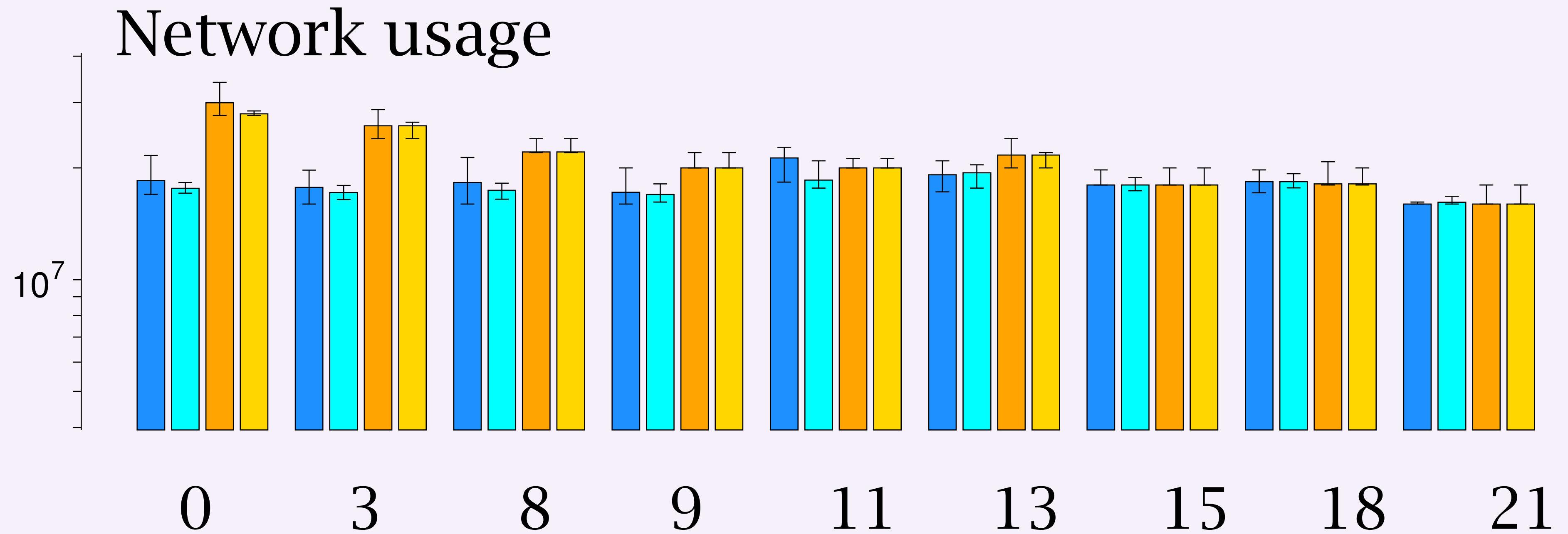


Tree (im)balance





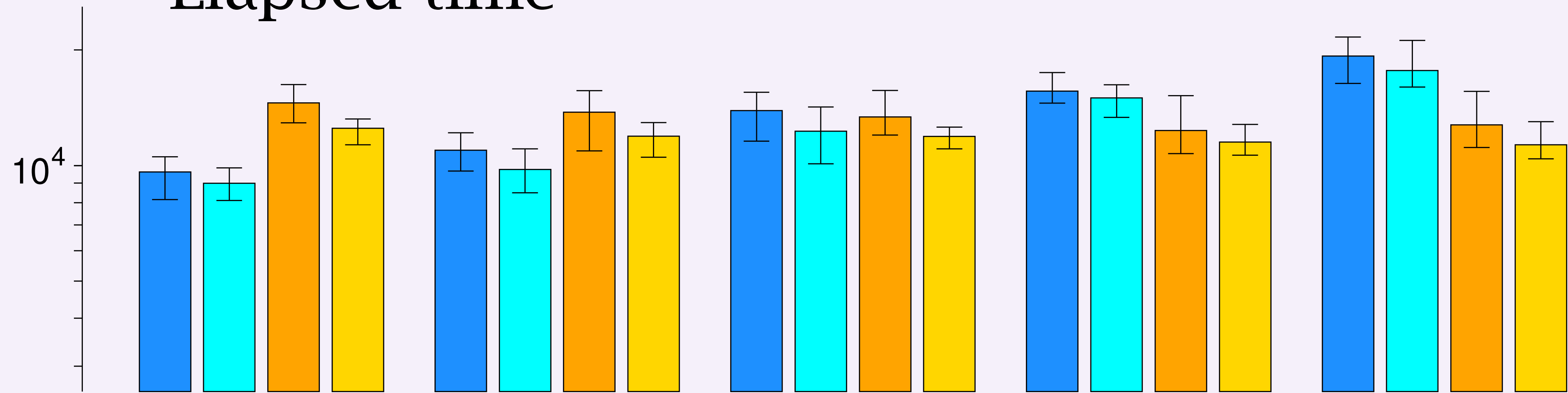
Tree (im)balance



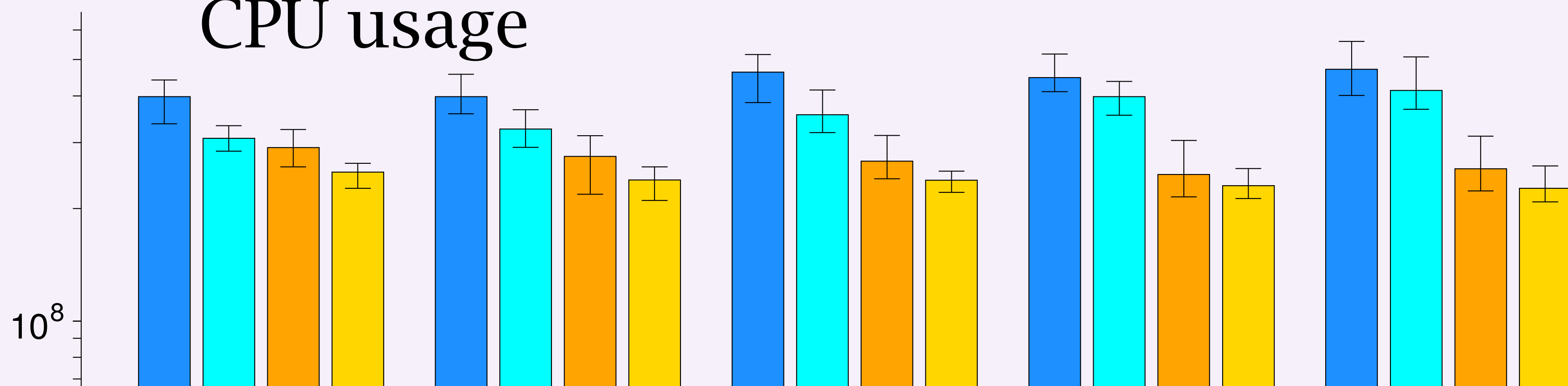
Workload balance

- Perfect availability, λ=10000 steps
- Server C evaluates a 3-way join between itself, A and B
- Output of A and B fixed, vary their relative workloads

Workload balance



5:5 6:4 7:3 8:2 9:1



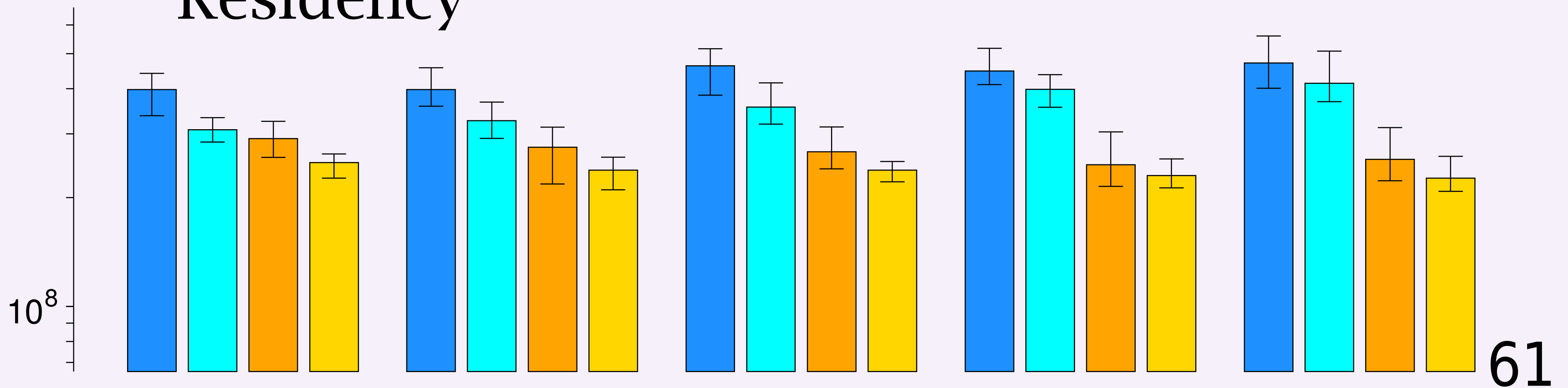
Workload balance

Footprint

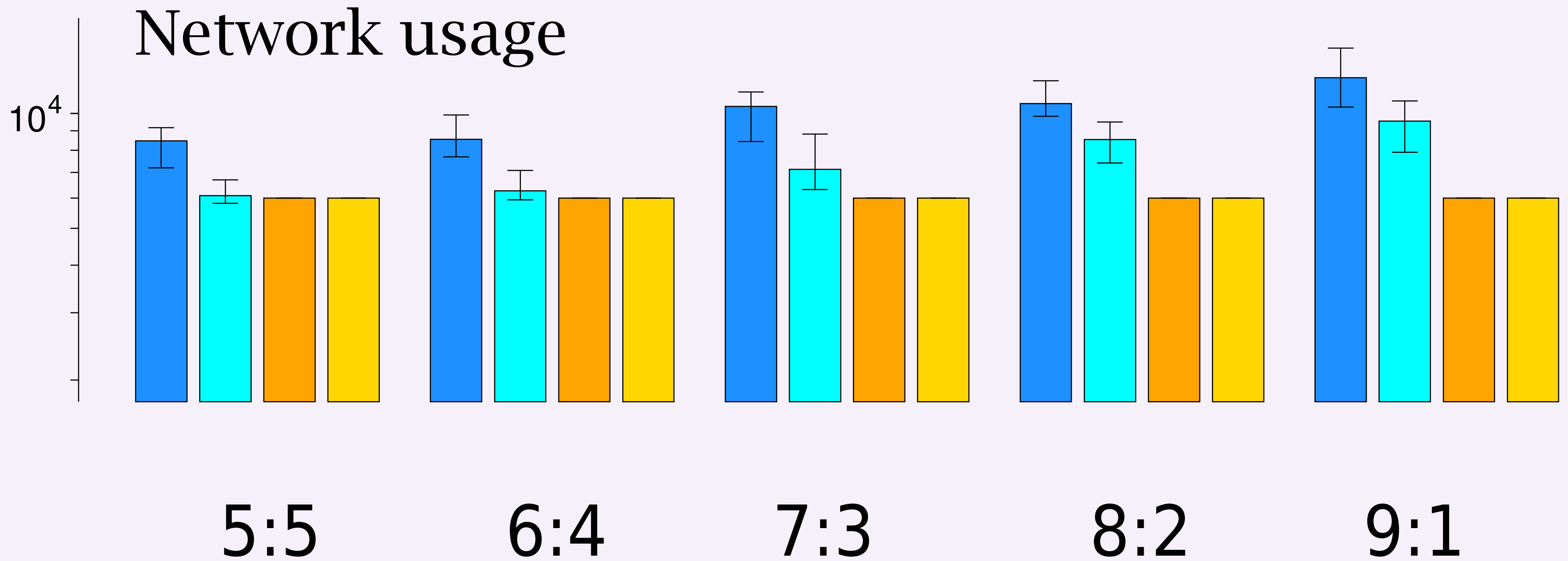
10^{10}
10^{9}
10^{8}
10^{7}

5:5 6:4 7:3 8:2 9:1

Residency



Workload balance



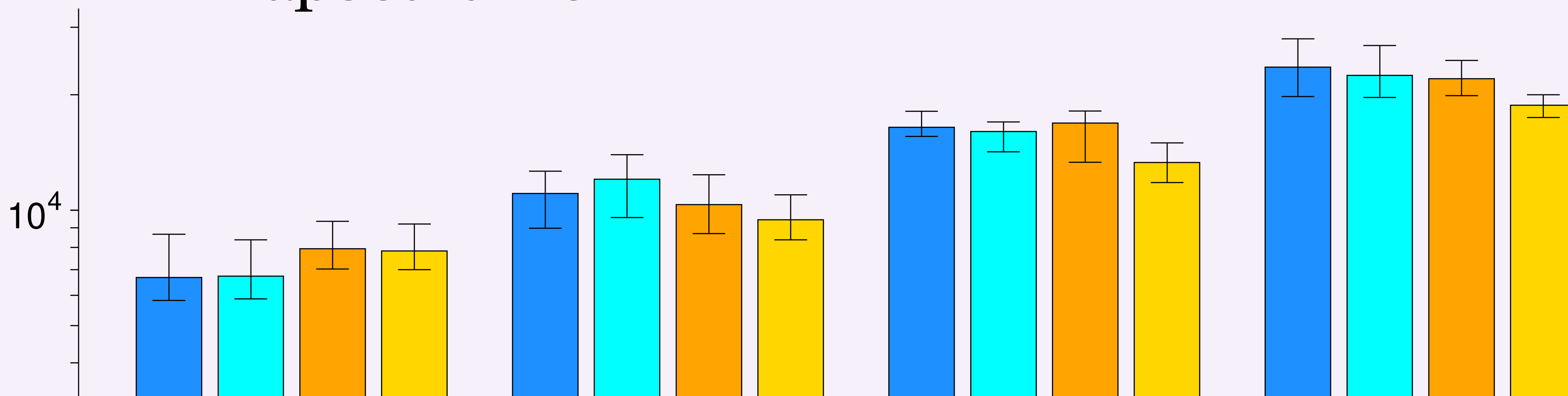
5:5 6:4 7:3 8:2 9:1

Fragmentation

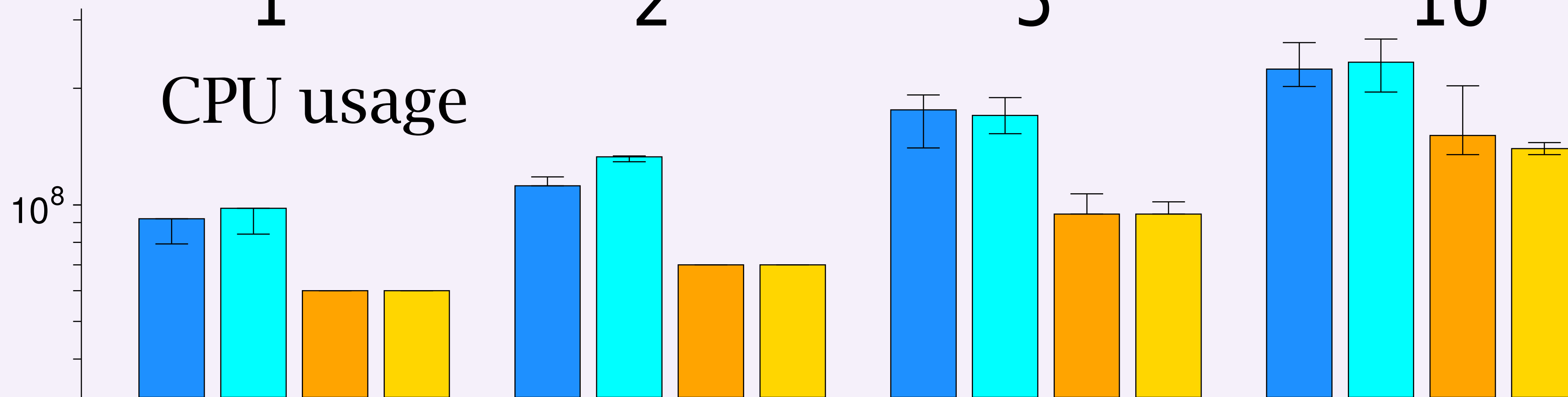
- a=u=5000 steps, λ=20000 steps
- Simple 3-way join: $(A \bowtie B) \bowtie C$
- B fixed size, variable number of fragments: 1, 2, 5, 10

Fragmentation

Elapsed time

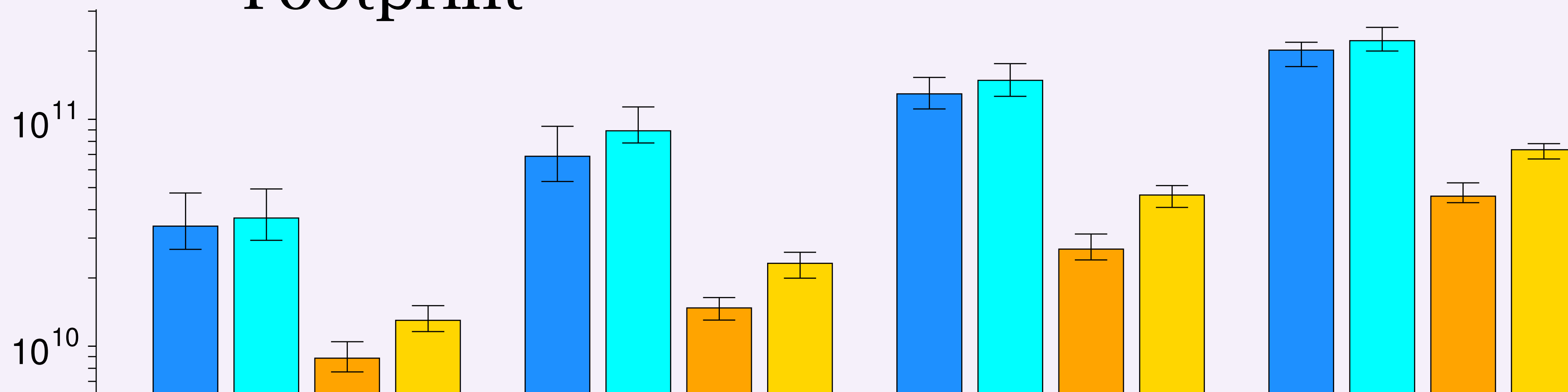


1 2 5 10

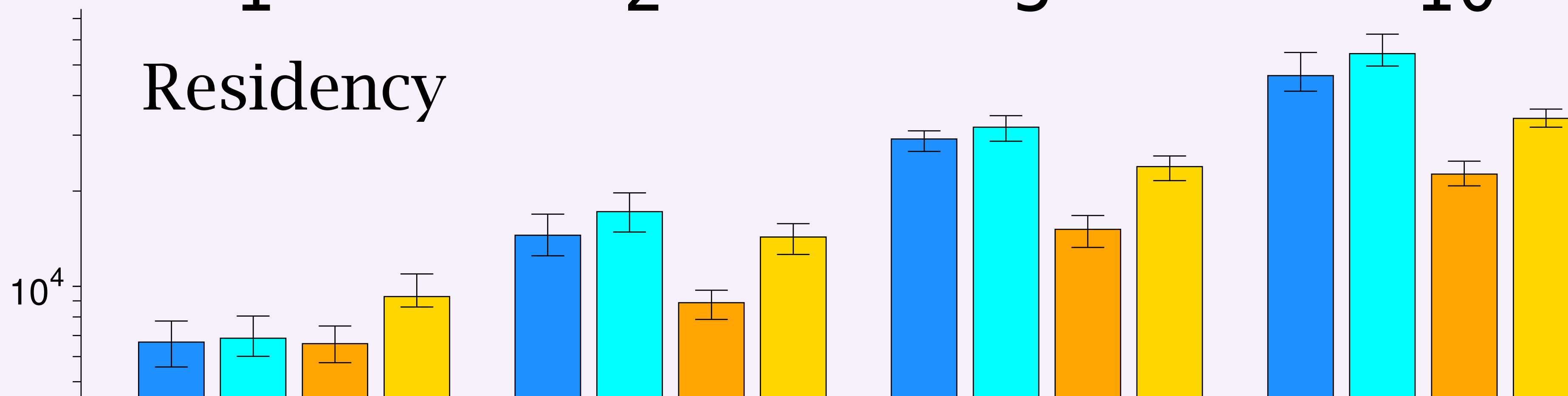


Fragmentation

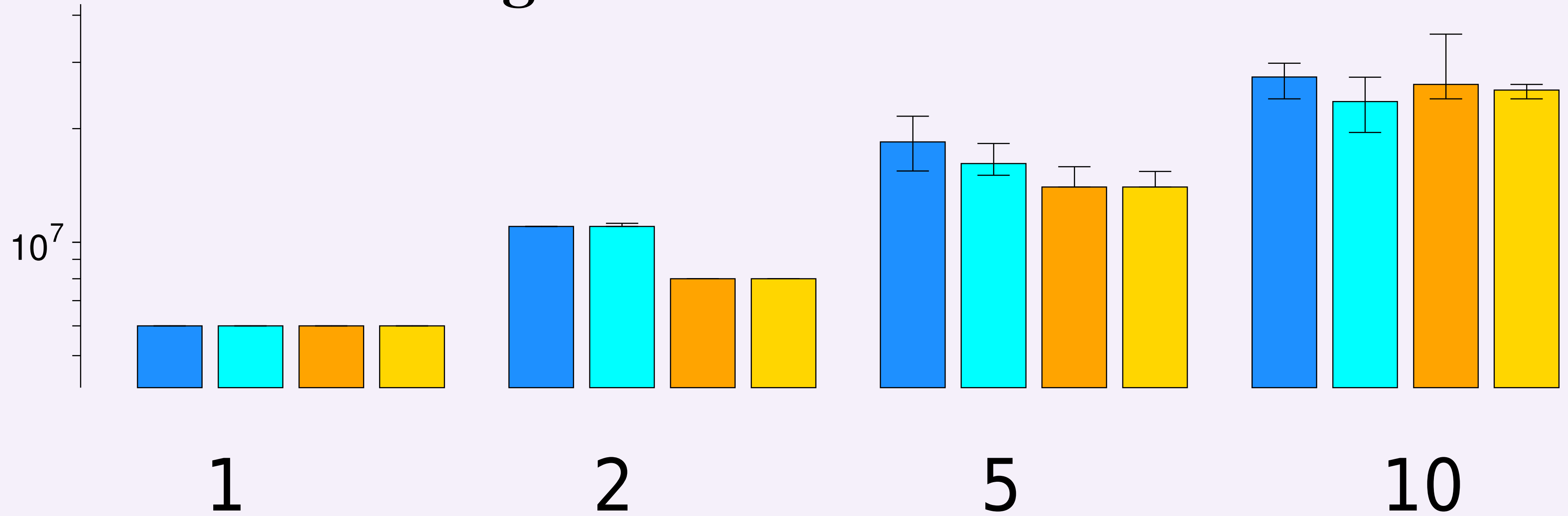
Footprint



1 2 5 10



Fragmentation



Summary

- Undependable environments (availability, terminations), favor MQPs
- Complex queries (join depth, # of fragments), favor MQPs
- Balance (tree/workload) favors pipelining; Imbalance favors MQPs
- MQPs often have worse elapsed times and network usage, but usually have much better residency, footprint, CPU usage

Conclusions

- Pipelined evaluation is a complicated multi-phase multi-server interaction
- Mutant evaluation is a sequence of simple point-to-point requests
- Tradeoff: much better throughput for slightly worse latency
- Distributed querying does not have parallel querying!

Related work

- Distributed DBMS research since the '70s!
  - Less interest in industry compared to parallel DBMS/clusters
  - Trend towards loosely-coupled, federated architectures
  - Transactional guarantees are hard
  - Freshness/Currency vs. Consistency
  - Schema integration is hard

Related work (2)

- Mariposa
- Parachute Queries
- d3log & Intensional Answers
- Alternative query evaluation methods
  - Referral
  - Leasing
  - Chaining
  - Publish-Subscribe

Related work (3)

- The Web!
  - Web Services, SOAP, UDDI
  - REST
- ActiveXML
- P2P & Distributed Hashtables

Extensions/Future work

- Distributed Catalogs
- MQP caching
- Result parking
- Pipelined embedding
- MQP strains & streams
- Metadata piggybacking
- Meta-level processing: load balancing, replication control,triggers etc.

Conclusions

- Pipelined evaluation is a complicated multi-phase multi-server interaction
- Mutant evaluation is a sequence of simple point-to-point requests
- Tradeoff: much better throughput for slightly worse latency
- Distributed querying does not have parallel querying!