Forecasting the Cost of Processing Multi-join Queries via Hashing for Main-memory Databases

Feilong Liu and Spyros Blanas

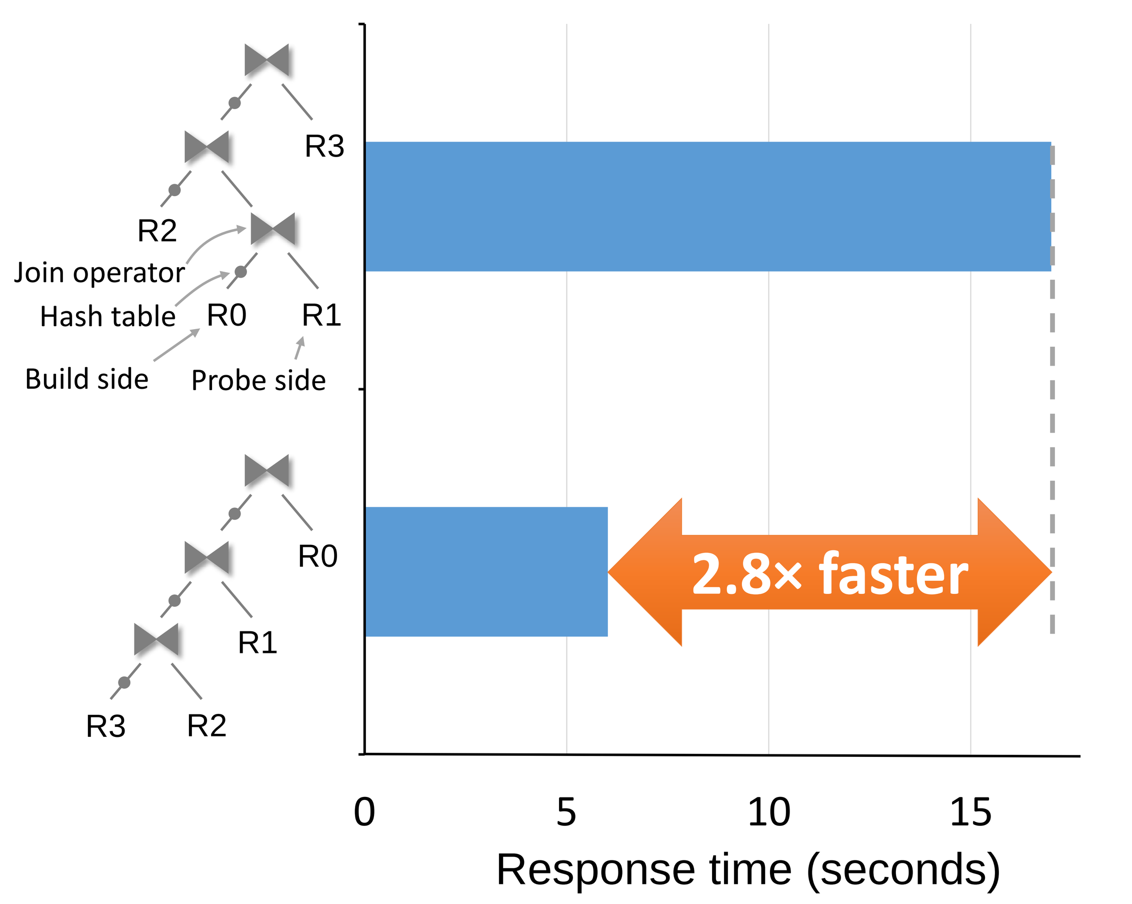
The Ohio State University

Key contribution

A cost model that accurately predicts the response time of ad-hoc SQL queries with multiple hash-based joins on an in-memory database

Why is modeling necessary?

Query	`SELECT SUM(R0.a + R3.b) FROM R0, R1, R2, R3 WHERE R0.b=R1.a, R1.b=R2.a, R2.b=R3.a`
In-memory database	Primary key-foreign key join between 4 tables: |R0|=32GB, |R1|=8GB, |R2|=2GB, |R3|=512MB



Different query plans produce the same output, but can have very different response time

Is a disk I/O model good enough?

Current approach: predict the response time of different query plans using a disk I/O model

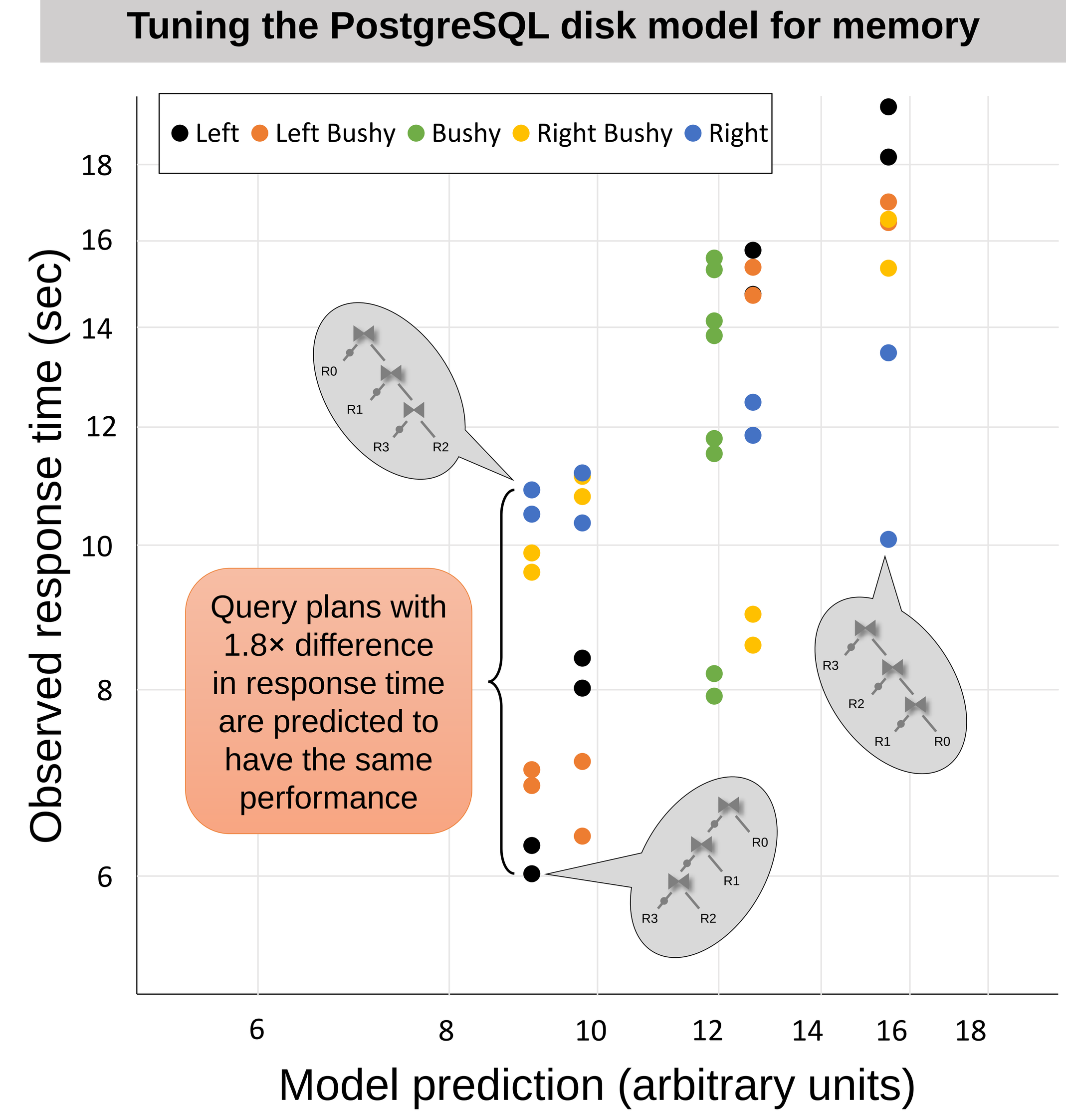
- Each disk access is classified as either a *sequential* access (n_s) or a *random* access (n_r)
- Each access type is assigned its own cost c_s or c_r

$$Cost(Q) \propto n_s \cdot c_s + n_r \cdot c_r$$

Tuning for an in-memory setting:

- We use the PostgreSQL query optimizer and statistics to obtain n_s and n_r
- With the observed response time from experiments, we use linear regression to compute optimal costs c_s and c_r

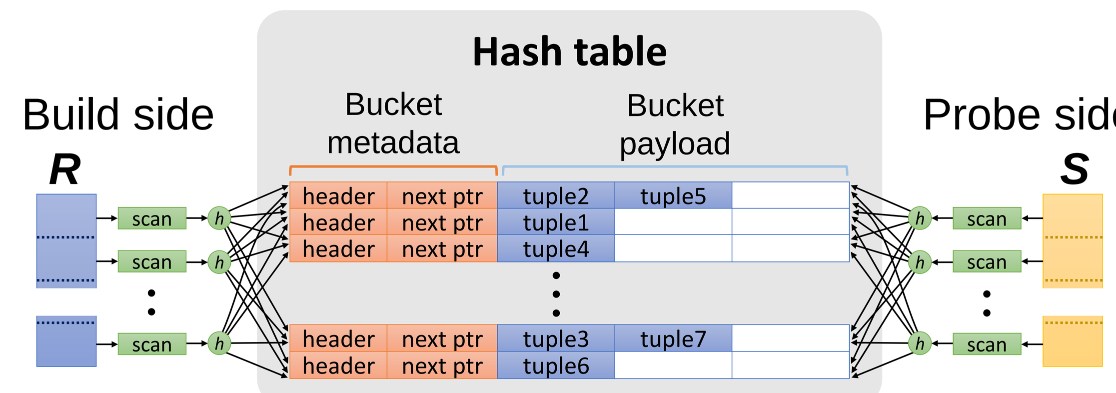
Tuning the PostgreSQL disk model for memory



It is not sufficient to tune traditional disk I/O models for main memory

Background: single join algorithm

Non-partitioned in-memory hash join



Our memory I/O model

We develop a memory I/O model to predict the response time of different hash-based multi-join query plans on an in-memory database

Our thesis: Response time is dominated by the cost of accessing main memory

- Each memory access is classified into one of the four types:

SR	Read one cache line sequentially
RR	Read one random cache line
SW	Write one cache line sequentially
RW	Write one random cache line

- For every access type, the model computes the number of accesses $N(SR)$, $N(RR)$, $N(SW)$ and $N(RW)$
- Each access type is assigned its own weight w_{SR}, w_{RR}, w_{SW} and w_{RW}

$$Time(Q) \propto w_{SR} \cdot N(SR) + w_{RR} \cdot N(RR) + w_{SW} \cdot N(SW) + w_{RW} \cdot N(RW)$$

Computing the weight $w(\cdot)$:

- We run microbenchmarks to calculate the relative cost of each type of memory access

Calculating the number of accesses $N(\cdot)$:

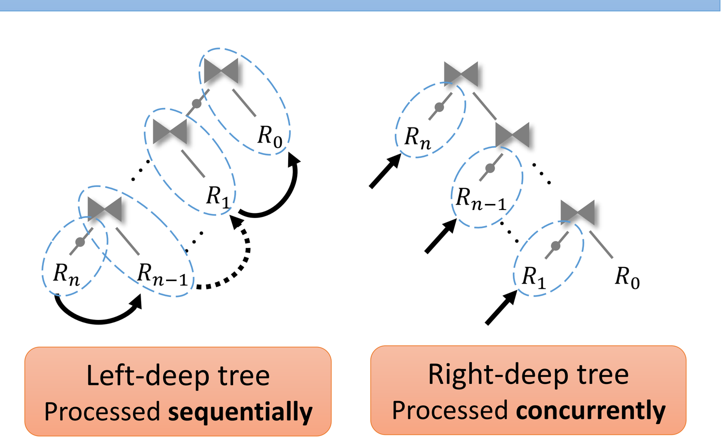
- Only memory accesses leading to a last level cache miss are taken into account; the model is oblivious to the multi-level cache hierarchy and any NUMA effects
- The cardinality of the intermediate join results is assumed to be known
- The memory access count of a query plan is the sum of the memory access counts of all operators
- We model the build and probe phases of a join operation separately
- In the hash join build phase, inserting into the hash table will cause *RW* and *SW* activity
- In the hash join probe phase, probing the hash table will lead to *RR* and *SR* activity

See paper for formulas

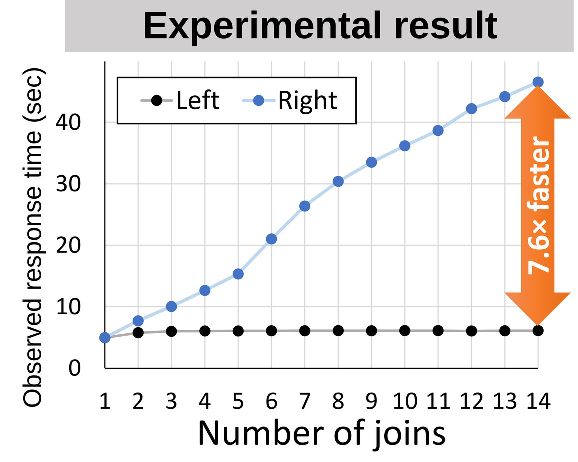
Left-deep tree vs. right-deep tree

Prior work in parallel databases advocates **right-deep** trees:

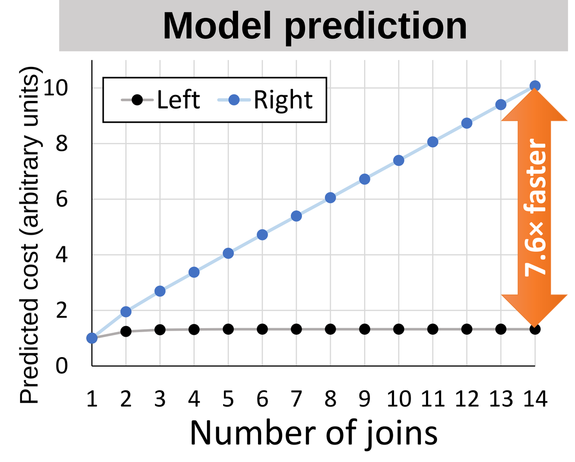
- Hash tables can be built concurrently
- Largest table is fed to a single probe pipeline



Experimental result



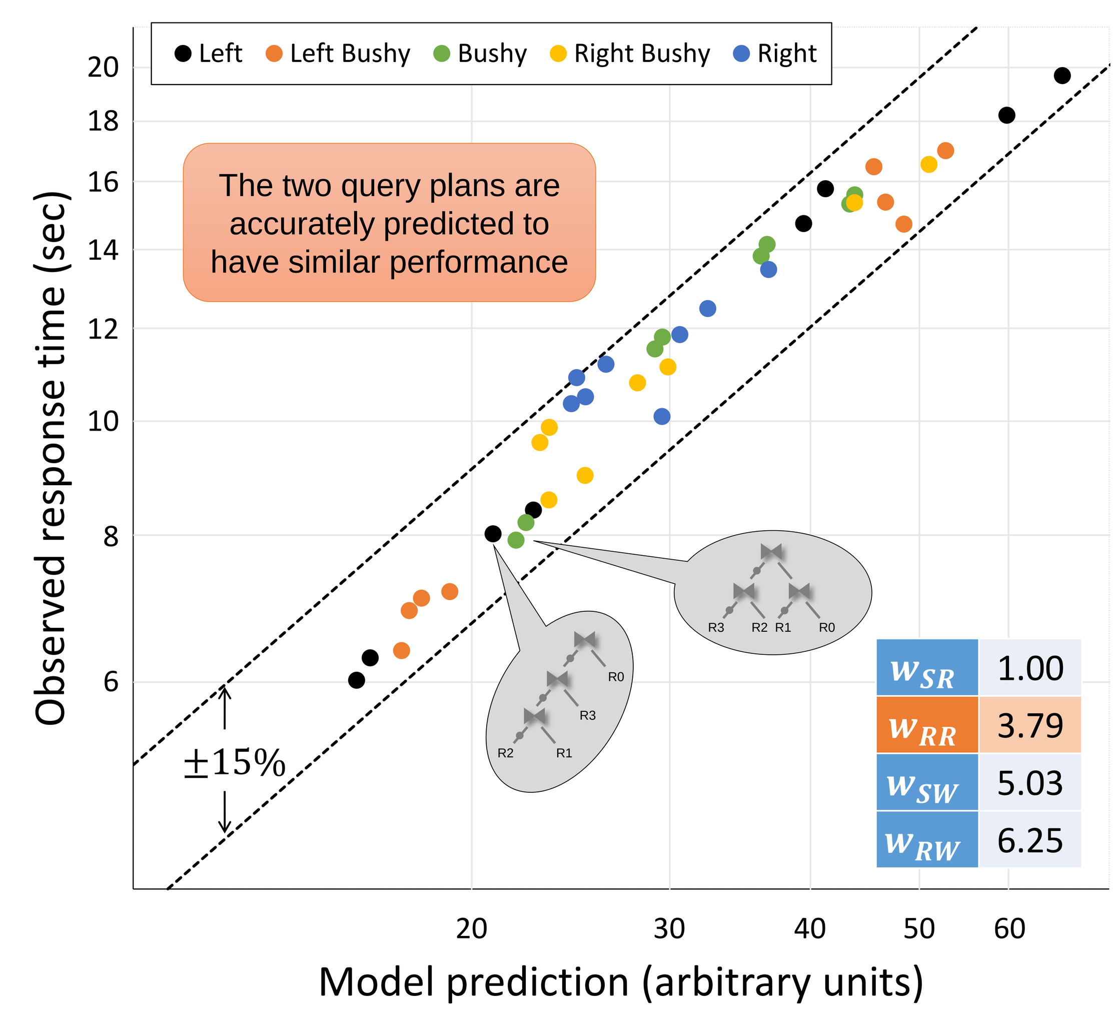
Model prediction



Our model corroborates that the optimal left-deep tree can be **8× faster** than the optimal right-deep tree for queries with more joins

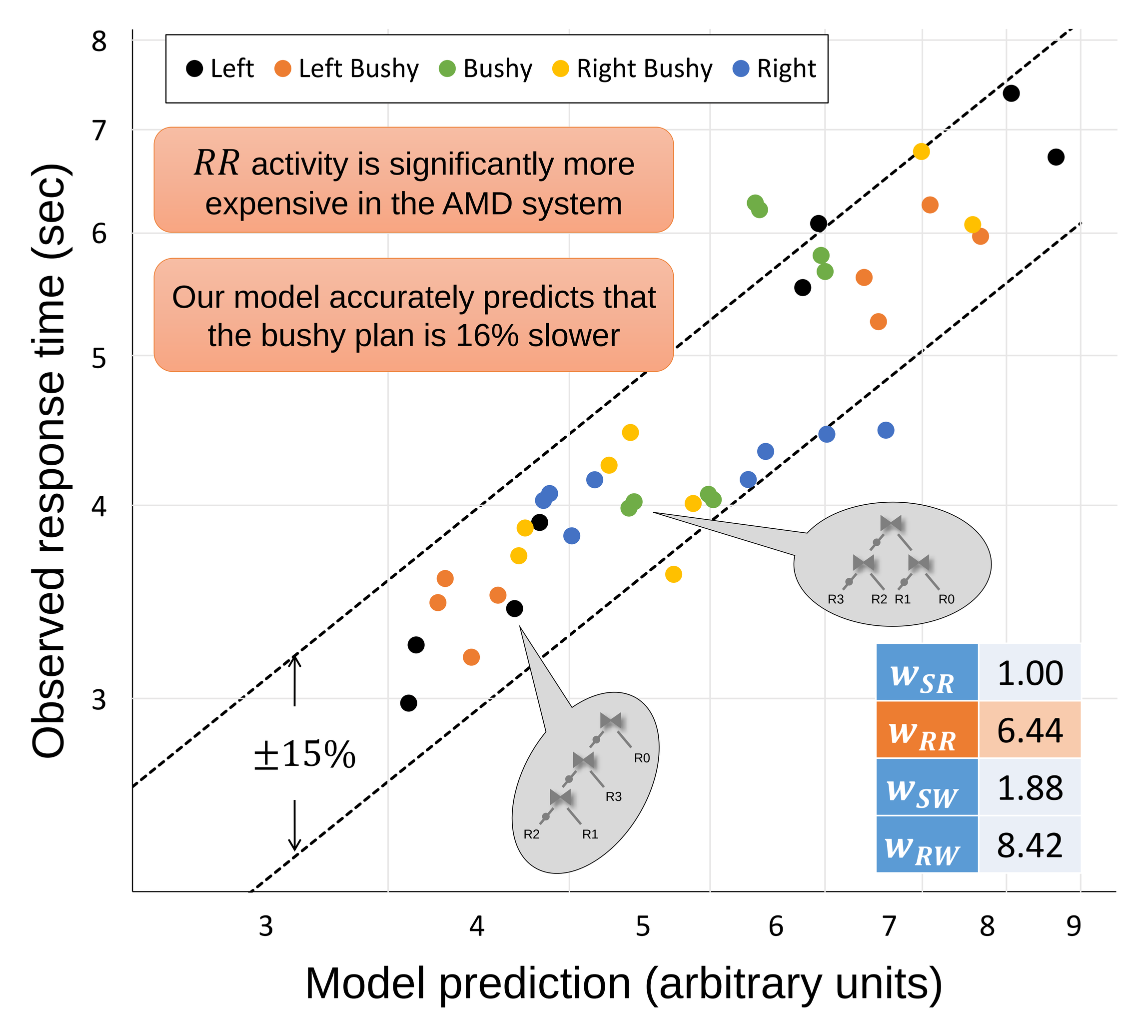
Adaptability to different hardware

Intel Xeon E5, 2 NUMA nodes, 24 cores



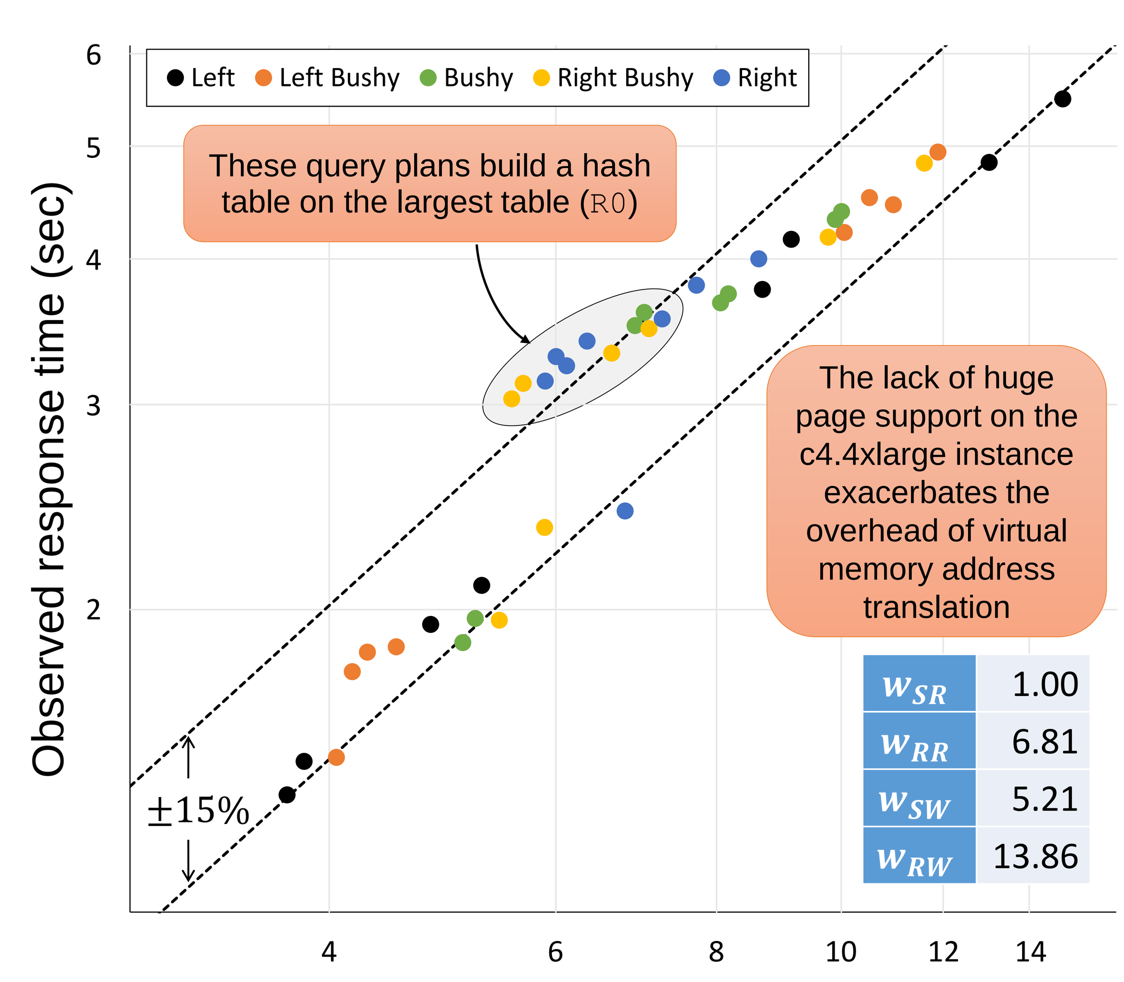
w_{SR}	1.00
w_{RR}	3.79
w_{SW}	5.03
w_{RW}	6.25

AMD Opteron, 4 NUMA nodes, 24 cores



w_{SR}	1.00
w_{RR}	6.44
w_{SW}	1.88
w_{RW}	8.42

Amazon EC2 c4.4xlarge, 16 vCPUs



w_{SR}	1.00
w_{RR}	6.81
w_{SW}	5.21
w_{RW}	13.86

The proposed model accurately predicts response time and successfully adapts to different hardware

Conclusions

- Our model accurately predicts the memory access activity when evaluating ad-hoc multi-join queries
- For an in-memory database, the memory access cost is an accurate proxy for query response time
- Sequential join evaluation can avoid the cascading effect of cardinality estimation errors and is a viable in-memory query execution strategy