CONNECTED DATA

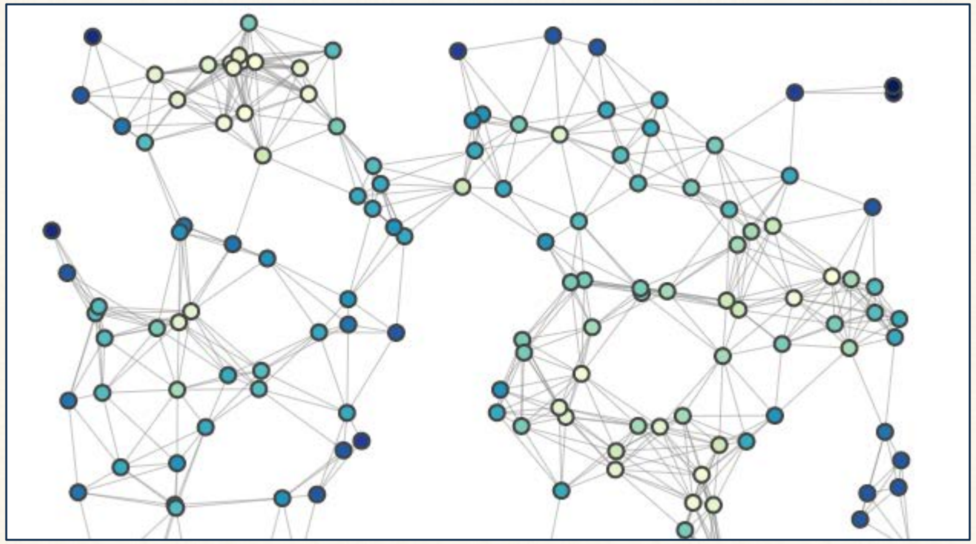
Pushing the Envelope of Data Management Systems

Marco Serafini

UNIVERSITY OF MASSACHUSETTS AMHERST

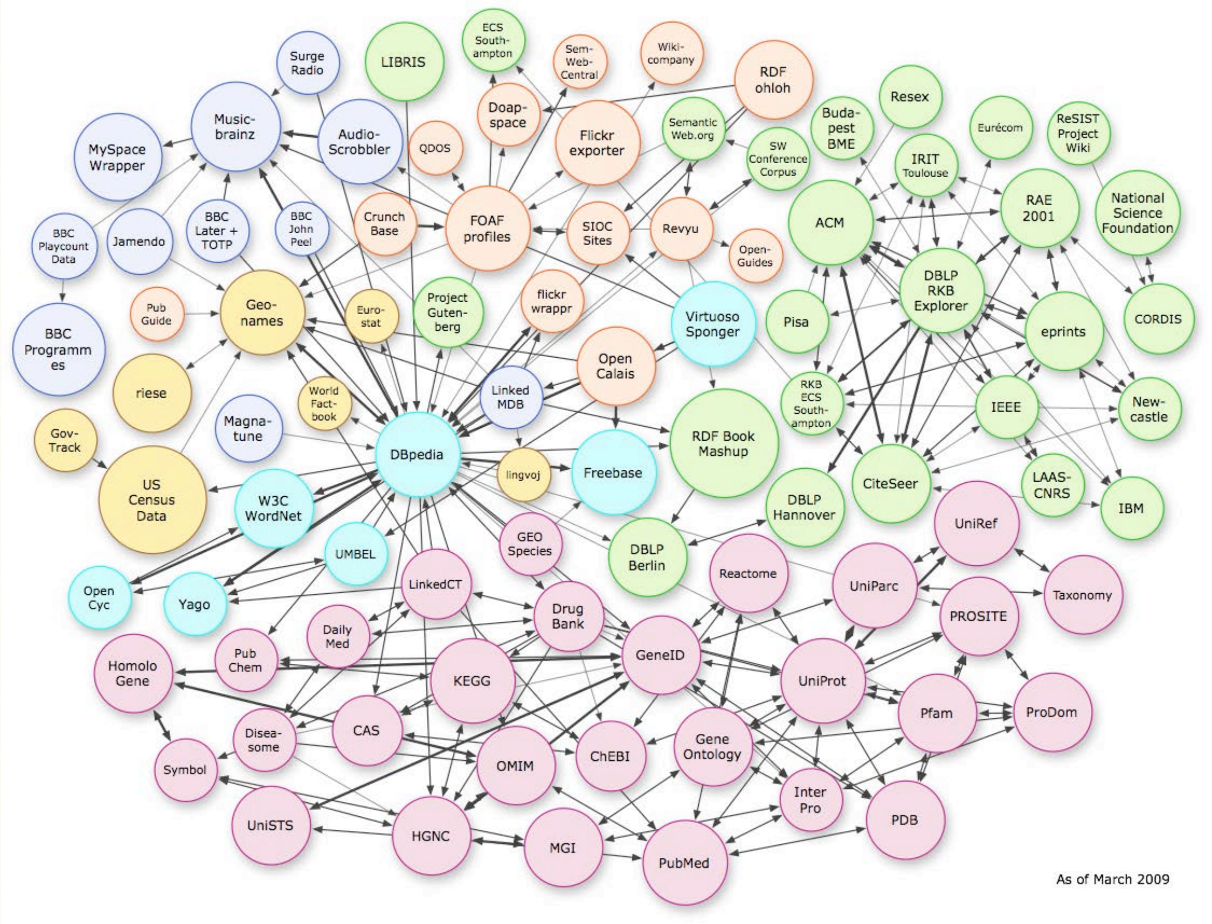
CONNECTED DATA

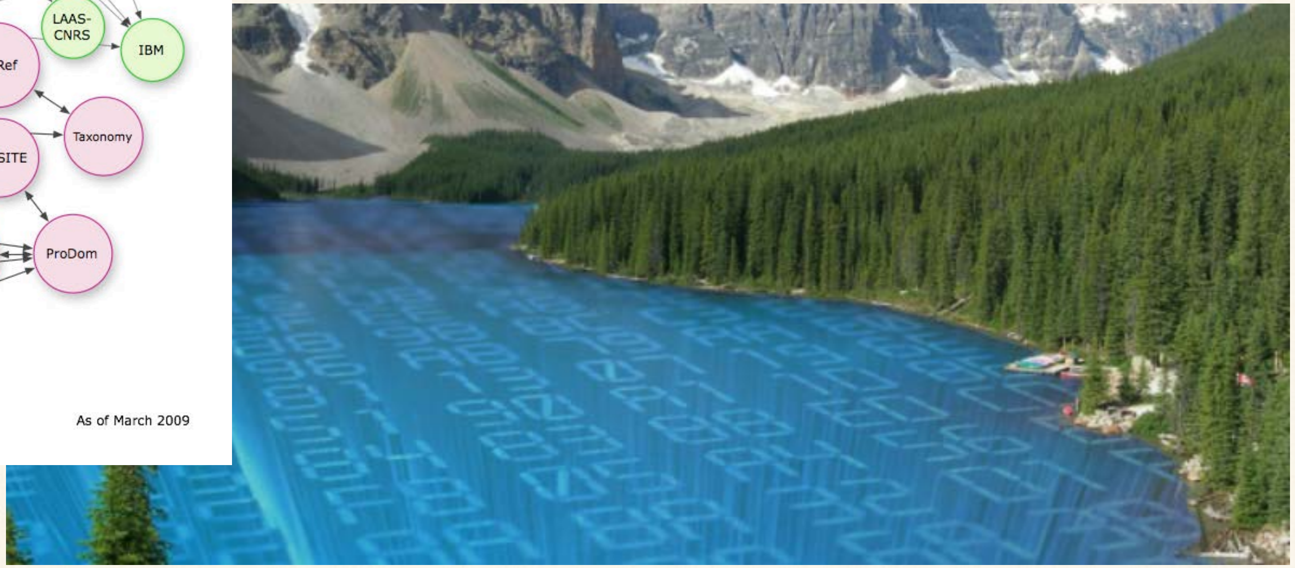
- Entities + Relationships
- Each entity can have an arbitrary number of relationships
  - Extreme skew: huge variance in number of relationships per entity
- Relationships are added on the fly



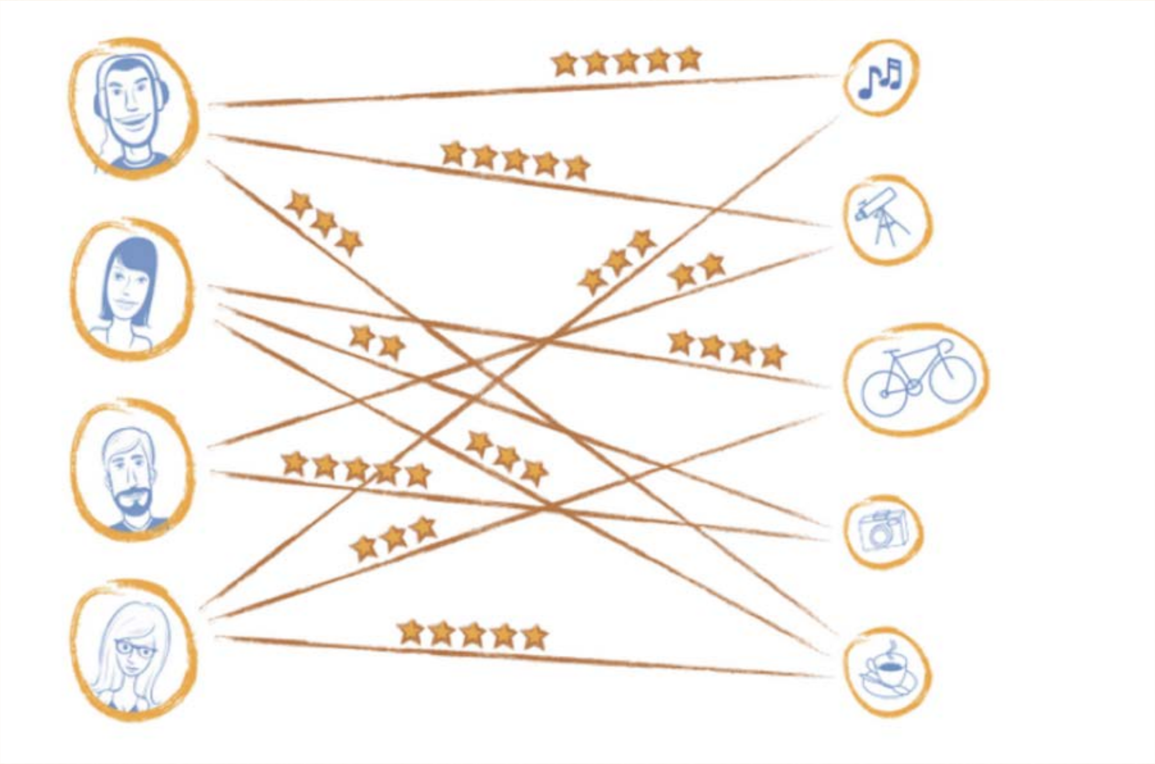
SOCIAL NETWORKS

KNOWLEDGE GRAPHS



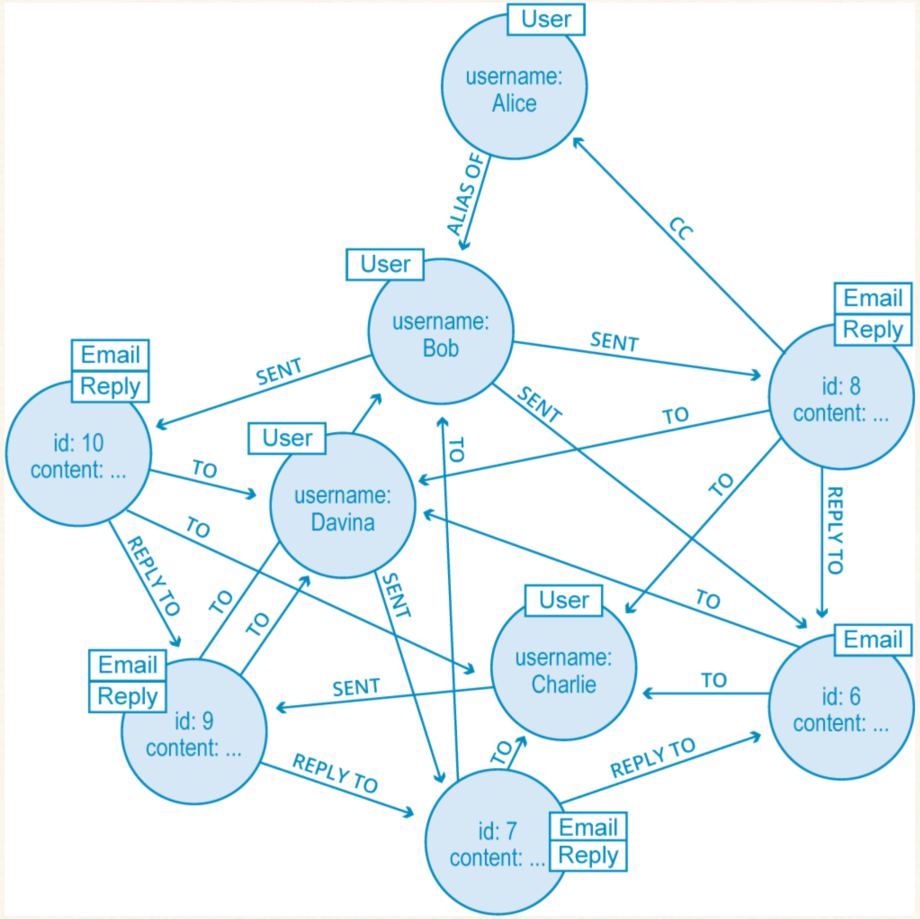


RECOMMENTADIONS & PERSONALIZATION



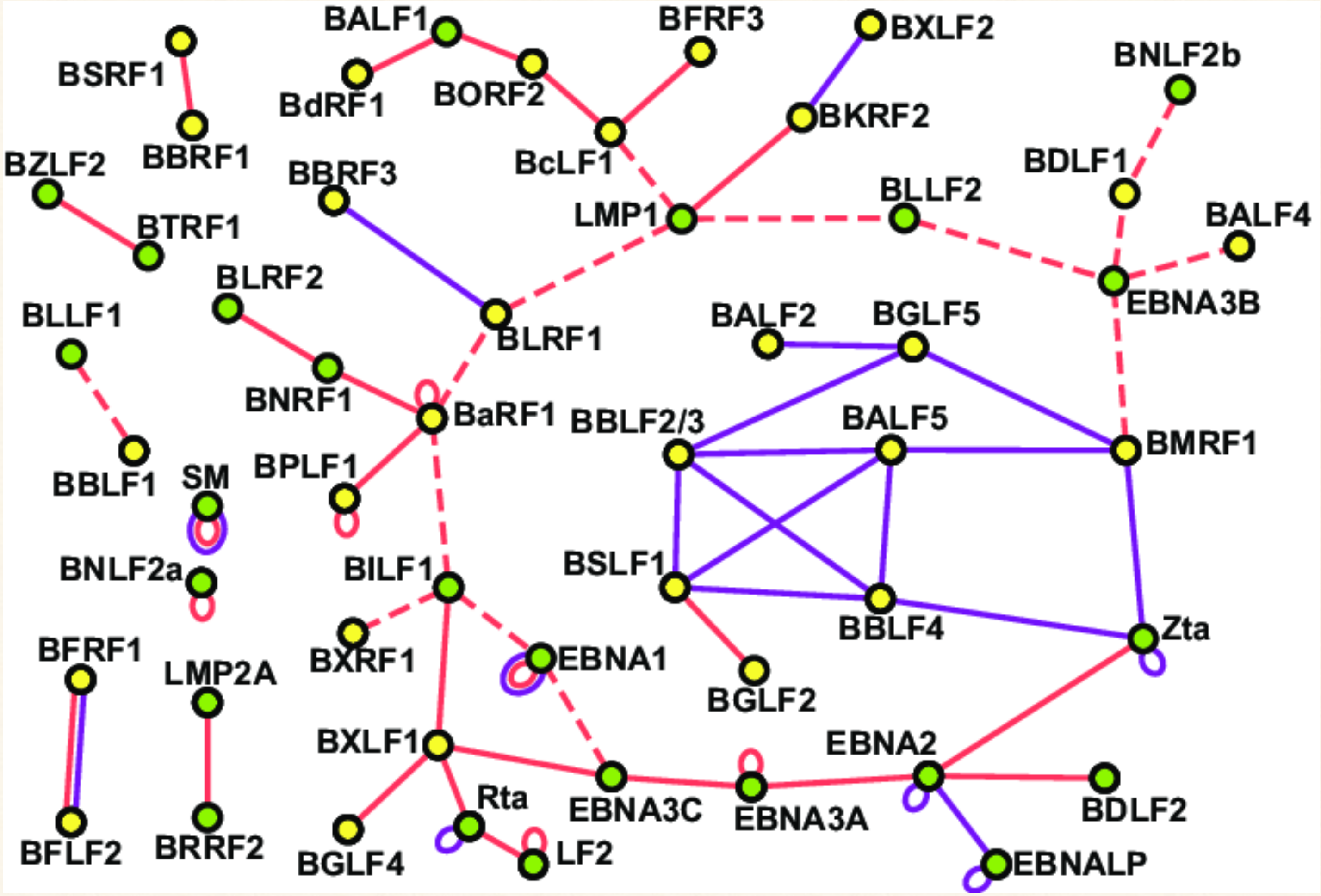
FINANCIAL DATA / FRAUDS

INFRASTRUCTURE/IoT MONITORING



DATA LINAGE / PROVENANCE

BIOLOGY



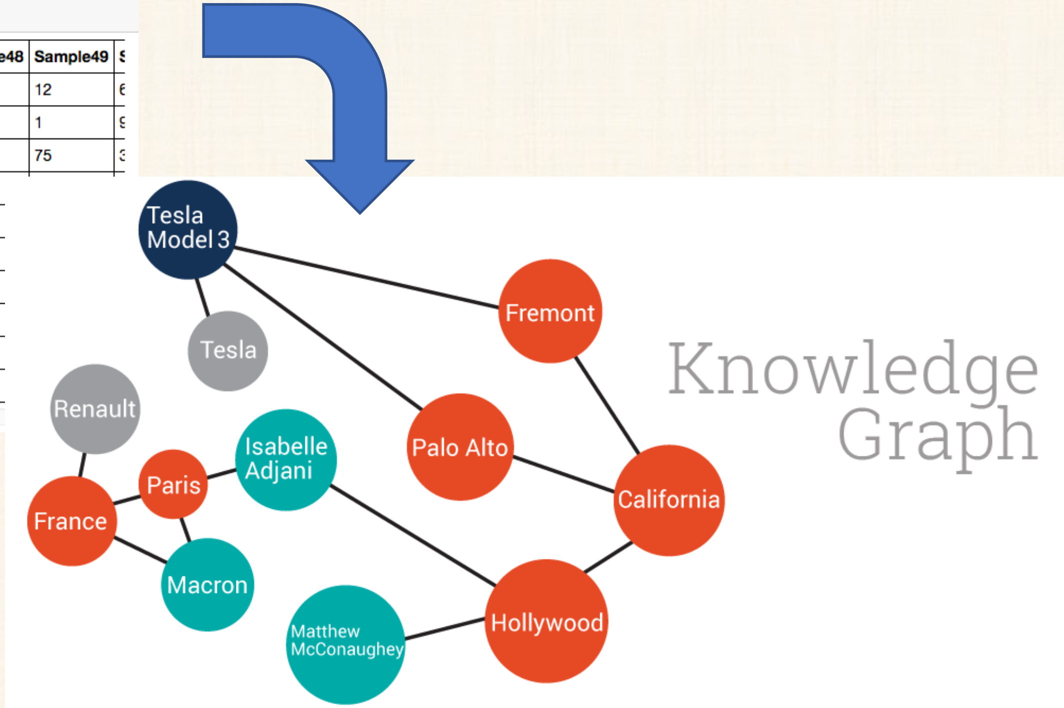
BSRF1
BBRF1
BALF1
BdRF1
BORF2
BFRF3
BcLF1
BXLF2
BKRF2
BNLF2b
BDLF1
BZLF2
BTRF1
BBRF3
LMP1
BLLF2
BALF4
EBNA3B
BLRF2
BLRF1
BLLF1
BNRF1
BaRF1
BALF2
BGLF5
BBLF2/3
BALF5
BMRF1
BBLF1
SM
BPLF1
BNLF2a
BILF1
BSLF1
BBLF4
Zta
BXRF1
EBNA1
BFRF1
LMP2A
BGLF2
BXLF1
EBNA2
Rta
EBNA3C
EBNA3A
BDLF2
BFLF2
BRRF2
BGLF4
LF2
EBNALP

LEARNING OVER CONNECTED DATA

- Leverage structural properties of data

```
In [3]: gLogCpmData = as.matrix(read.table(paste0(
            gWorkingDir, "heatmap_test_matrix.txt")))
        gLogCpmData
```

	Sample1	Sample2	Sample3	Sample4	Sample5	Sample6	Sample7	Sample8	Sample9	Sample10	⋯	Sample47	Sample48	Sample49	S
GeneA	30	67	34	98	32	3	79	15	6	18		97	49	12	6
GeneB	80	70	28	51	74	76	85	98	7	64		45	69	1	9
GeneC	43	36	41	24	71	76	91	50	81	57		21	10	75	3
GeneD	88	66	57	8	11	91	71	84	89	63		12	16		
GeneE	90	57	73	51	86	32	22	78	84	31		37	40		
GeneF	31	87	65	36	64	15	28	89	94	58		58	32		
GeneG	1	93	70	64	98	28	65	96	83	2		51	83		
GeneH	27	18	37	85	59	61	85	3	16	7		38	14		
GeneI	85	35	24	73	21	25	45	80	20	94		34	10		
GeneK	16	55	76	60	40	48	85	90	24	44		66	53		



MODELING CONNECTED DATA

GRAPH VS. RELATIONAL

GRAPH vs. RELATIONAL DATA MANAGEMENT

CONVENTIONAL WISDOM

"You should not reinvent the wheel"

"When you have a hammer everything looks like a nail"

A PRAGMATIC APPROACH

- It is **not** about graph vs. relational **data**
- It is about graph vs. relational **workloads**
  - Diverse applications and algorithms
  - Diverse data structures and APIs
- Graph DBMSs should **extend not reinvent**
  - Eventual convergence of implementations is possible and desirable

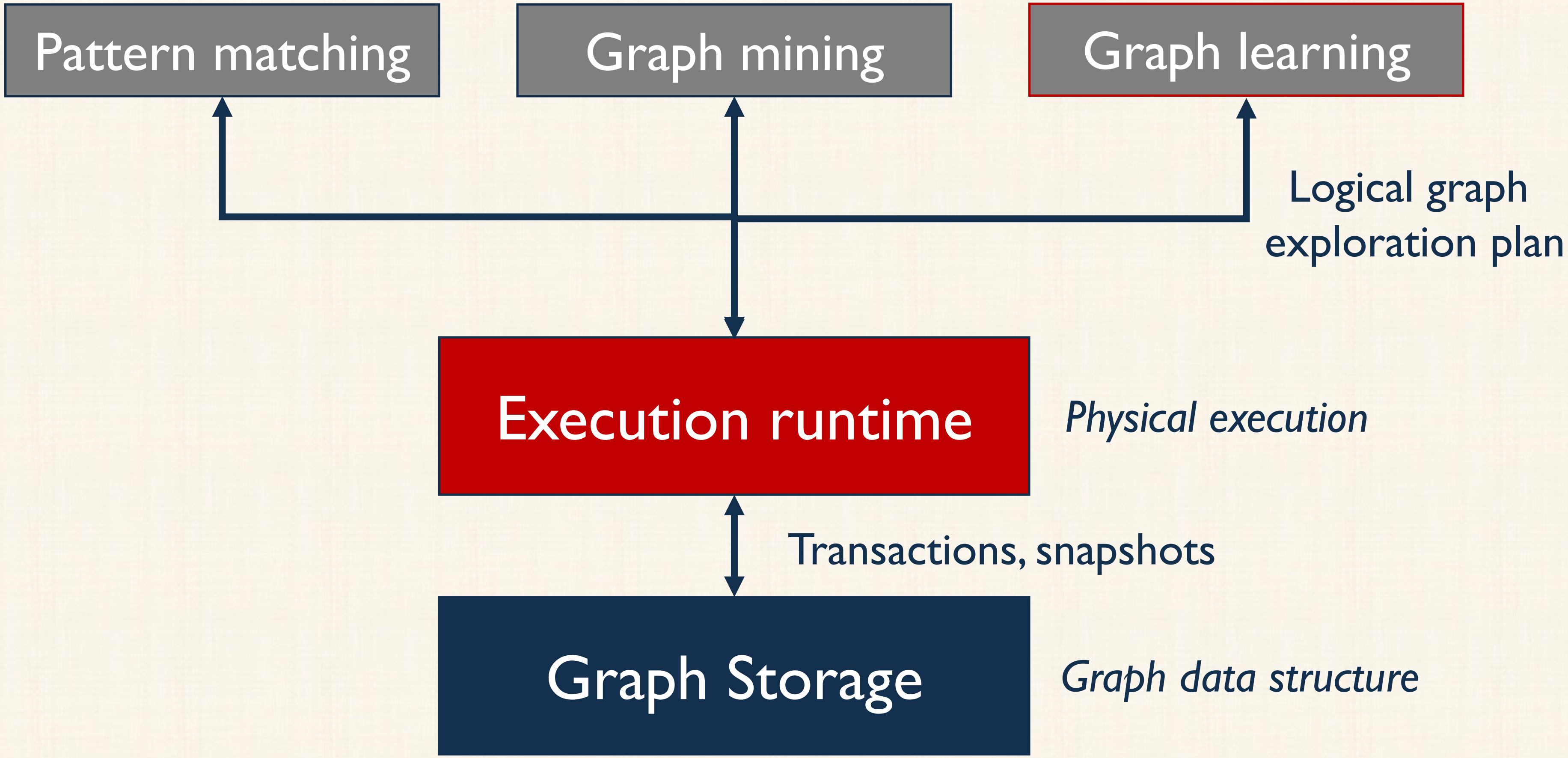
OPEN ISSUE: REAL-TIME

- **Real-time** analytics and queries on **dynamic** graphs
  - User likes product → gets real-time contextual recommendations
  - Failure/attack on system → immediate reaction
  - Fraud is attempted → blocked before financial loss
- Challenges
  - Graph algorithms are complex
  - Hopping edges requires random access
  - Sophisticated indexing, compression, and partitioning works only on read-only data

OPEN ISSUE: SCALE-UP ANALYTICS

- Advanced graph analytics are hard to scale out
  - Impossible to cleanly partition
- SIMD hardware offers massive scale-up parallelism
  - E.g. GPUs, Intel AVX, Intel Phi
- Challenge: hard to leverage SIMD for graph algorithms
  - Same problems as before: random access, poor caching, branching, …
  - But on an even larger scale

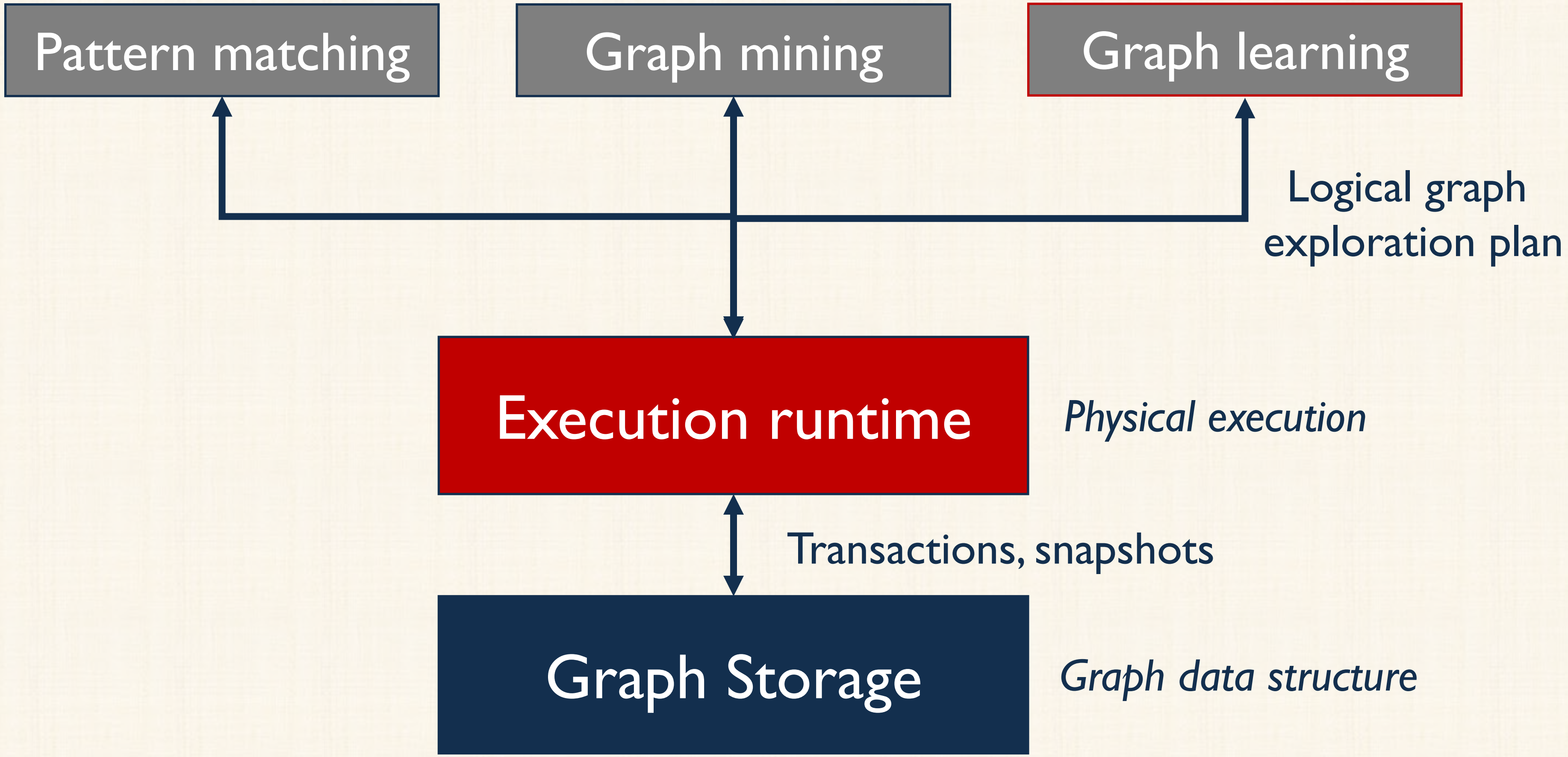
VISION


Pattern matching
Graph mining
Graph learning
Logical graph exploration plan
Execution runtime
Physical execution
Transactions, snapshots
Graph Storage
Graph data structure

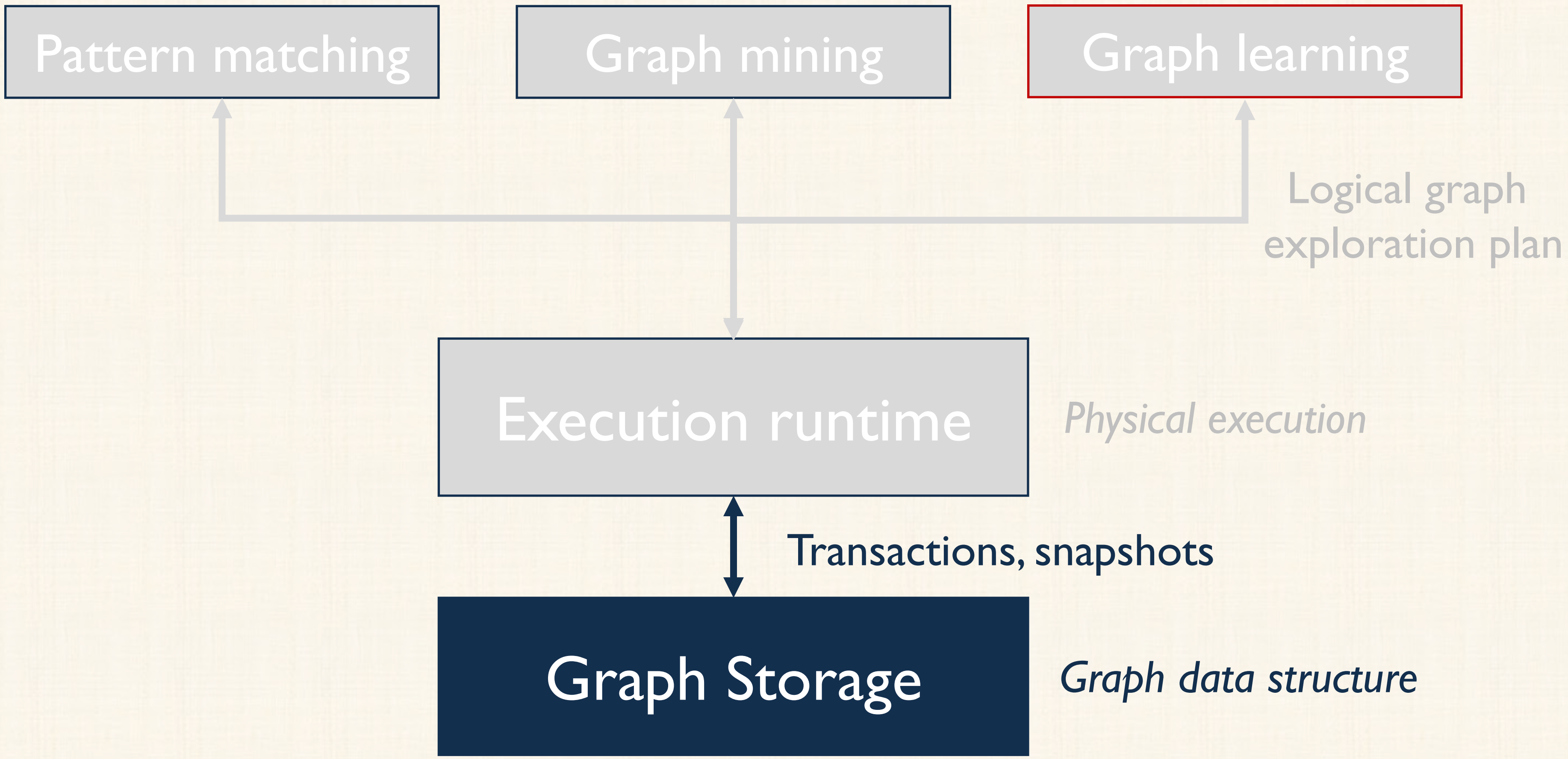
STORING CONNECTED DATA

AN EVOLUTIONARY APPROACH

VISION


Pattern matching
Graph mining
Graph learning
Logical graph exploration plan
Execution runtime
Physical execution
Transactions, snapshots
Graph Storage
Graph data structure

VISION


Pattern matching
Graph mining
Graph learning
Logical graph exploration plan
Execution runtime
Physical execution
Transactions, snapshots
Graph Storage
Graph data structure

RELATIONAL MODEL

- Connected data = dynamic relationships
  - New relationships among entities added all the time
  - Extreme skew: variance in # of relationships per entity
- Needed: flexible physical schema
  - Avoid frequent schema changes!
- Solution: Entity table + Relationship table

Entity ID	Properties

ENTITY (VERTEX)

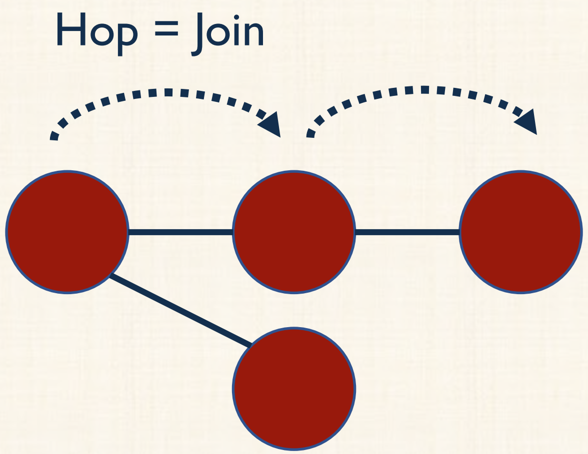
Source Entity ID	Destination Entity ID	Properties

RELATIONSHIP (EDGE)

WORKLOADS

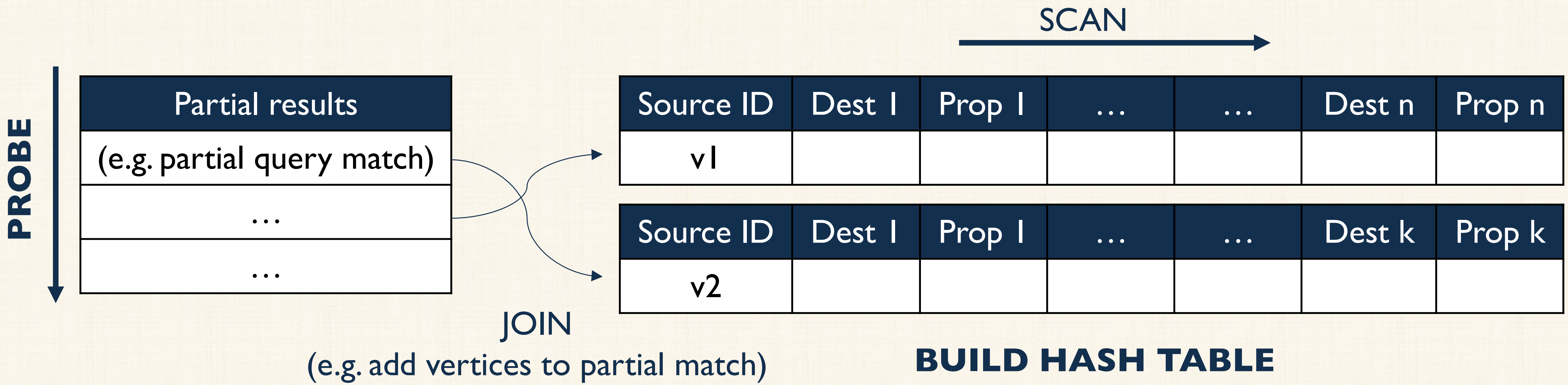
- Pattern/path based queries
  - Pattern queries
  - Reachability
  - Random walks
- Subgraph-based queries
  - Frequent subgraphs
  - Densest subgraphs
- Frontier-based queries
  - Shortest path
- Message passing
  - PageRank

Fundamental operation:
EDGE TRAVERSAL,
that is,
JOINS ON EDGE TABLE



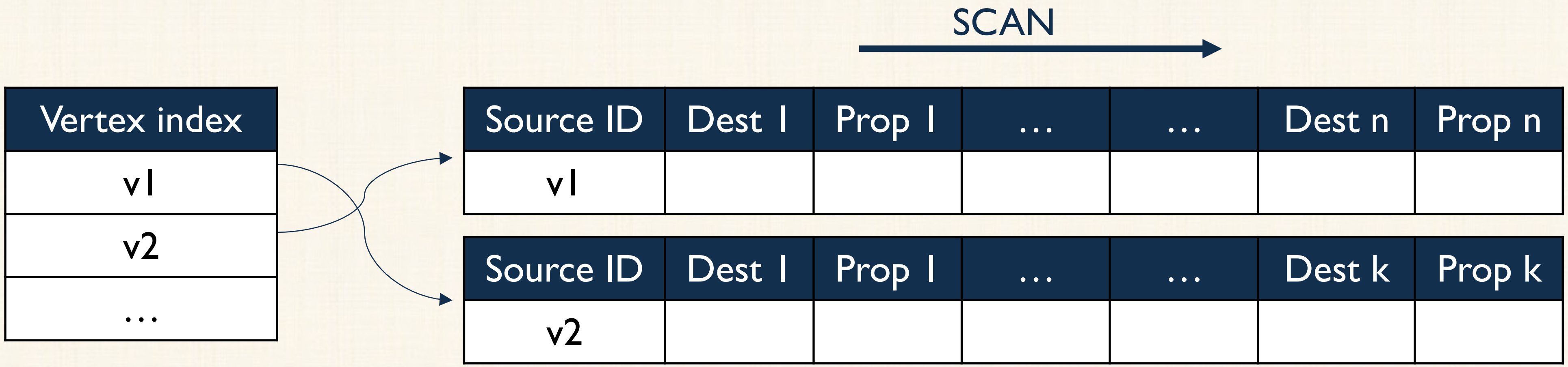
HASH-JOINING EDGE TABLE

- **Build:** hash table from edge table
- **Probe:** Scan through partial results and join/extend
- Typically, after the join scan (traverse) the joined edges



ADJACENCY LIST REPRESENTATION

- Adjacency lists = edge table optimized for joins
- **Graph storage systems**: optimized for adjacency lists

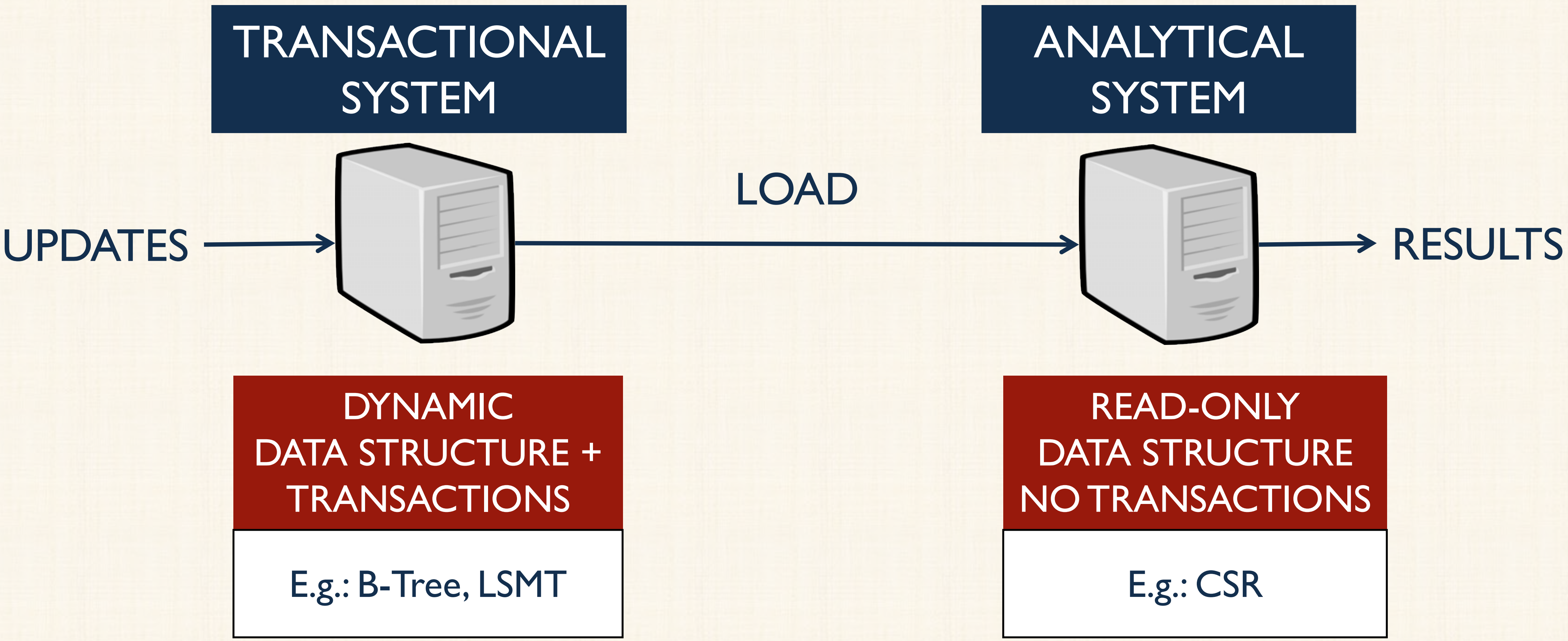


ADJACENCY LISTS

REAL-TIME WORKLOADS

- Real-time workloads
  - Dynamic data: Entities and relationships are added continuously
  - Queries and analytics on real-time data
- Examples: monitoring, real-time recommendations
- Graph storage requirements
  - Low-latency concurrent (transactional) updates
  - Low-latency reads from graph snapshots

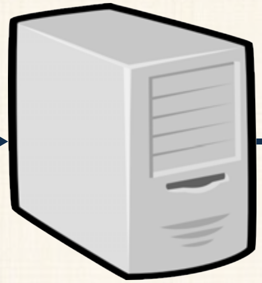
TYPICAL PIPELINE



LIVEGRAPH

REAL-TIME GRAPH
DATA MANAGEMENT

Updates
Real time queries
Snapshot queries



Results

TRANSACTIONAL
EDGE LOG

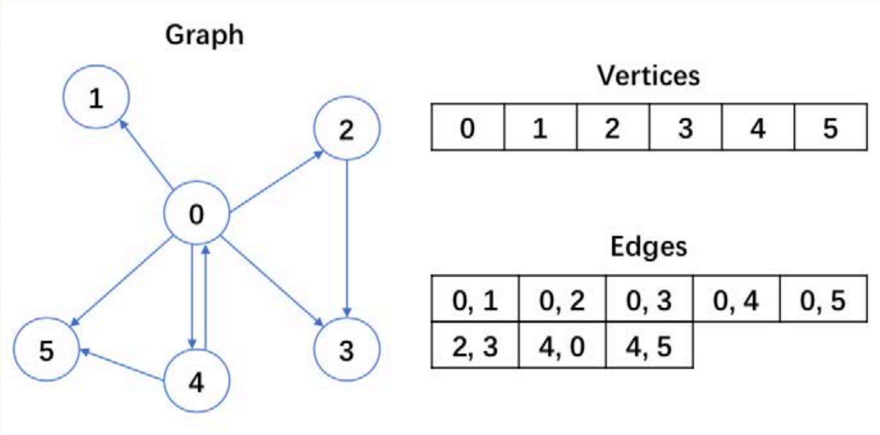
LIVEGRAPH

- Features
  - Embedded graph store
  - ACID transactions
  - Real-time reads on the live data (no data loading)
  - Snapshot isolation: wait-free reads
  - Multi-versioned (temporal/incremental queries)
- Key design choices
  - Sequential adjacency list scans
  - Fast insertions in constant time

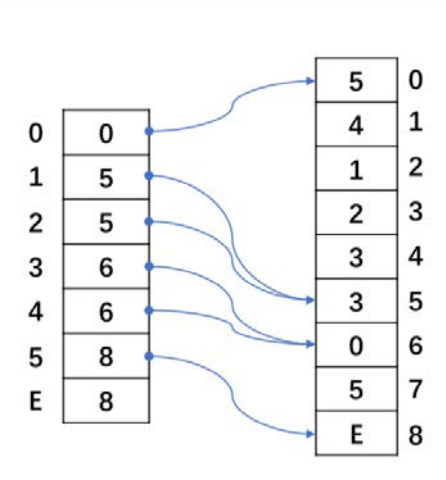
DATA STRUCTURE COMPARISON

- B+ Trees
  - LMDB
  - Typical RDBMS data structure
- Log-Structured Merge Trees
  - RocksDB
  - Skip-list + compressed runs
- Linked lists
  - Neo4J
- **Transactional Edge Log**

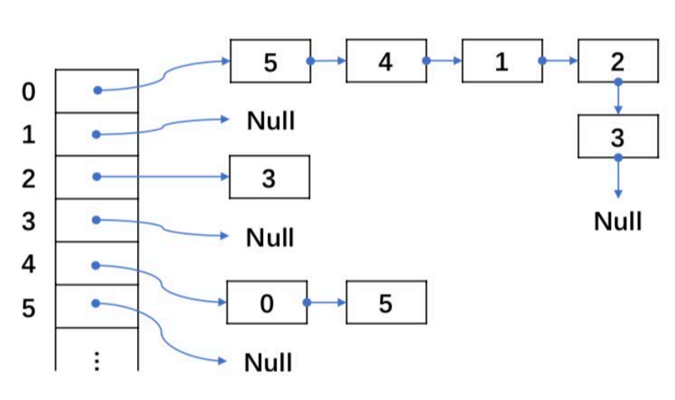
GRAPH REPRESENTATIONS



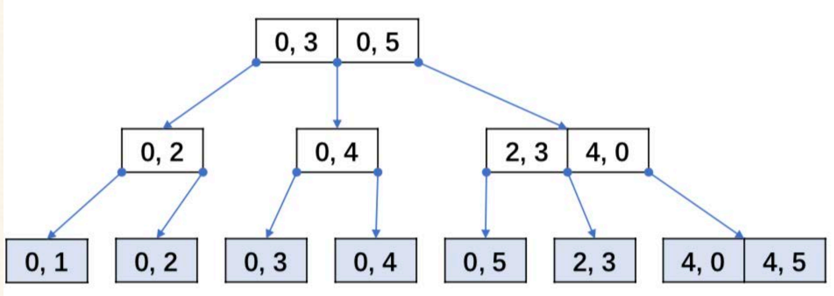
Input graph



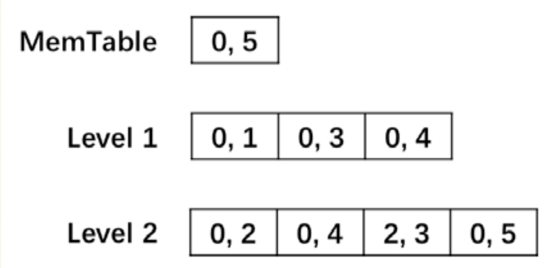
CSR (read-only)



Linked list



B+ tree

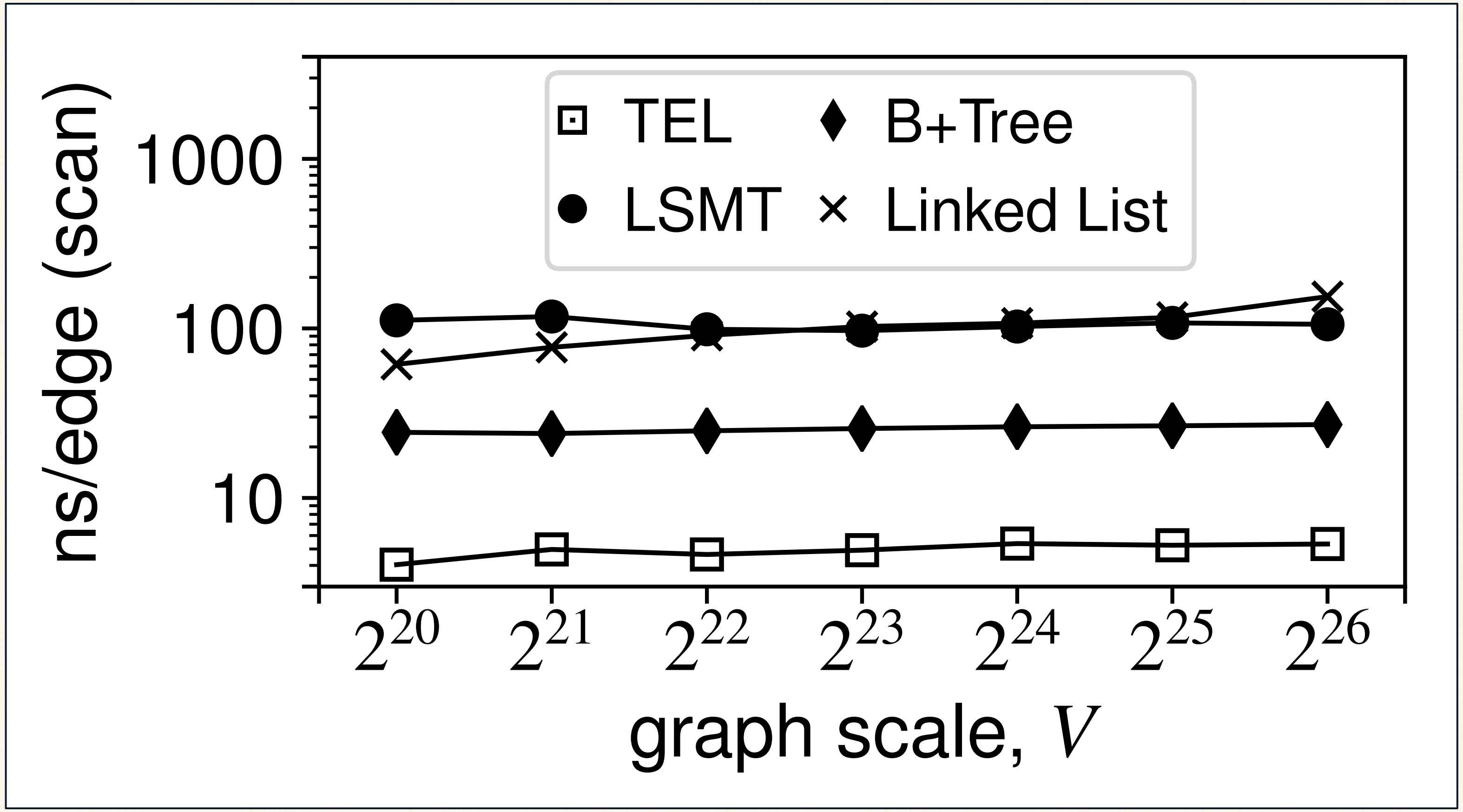


LSMT

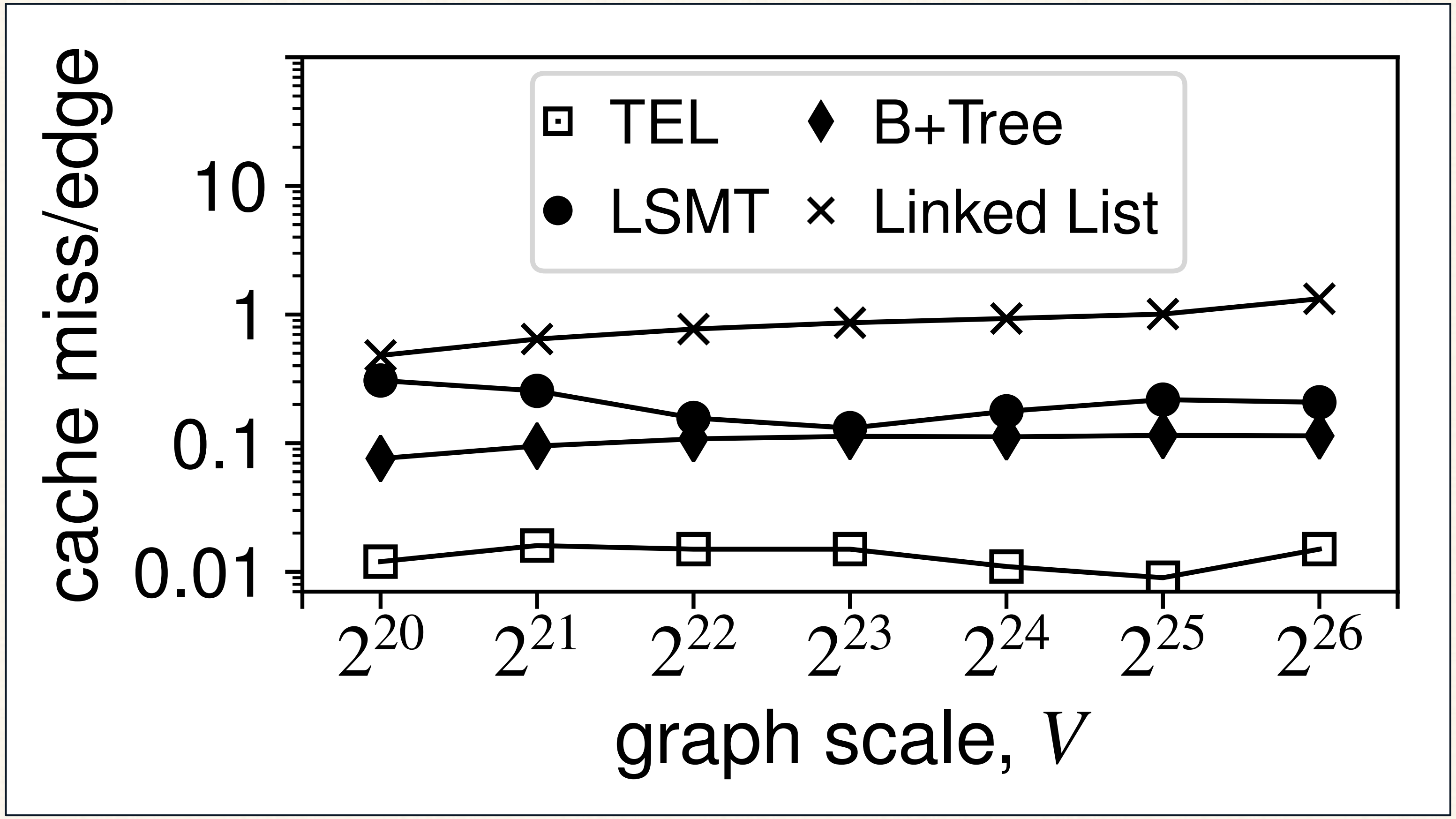
MICRO-BENCHMARK

- Seek & scan adjacency list
  - Seek: find adjacency list
  - Scan: get next edge in the adjacency list
- Data: Kronecker graph that fits in memory of one socket

EDGE SCAN


TEL
B+Tree
LSMT
Linked List
1000
100
10
ns/edge (scan)
2^{20}
2^{21}
2^{22}
2^{23}
2^{24}
2^{25}
2^{26}
graph scale, V

CACHE MISSES

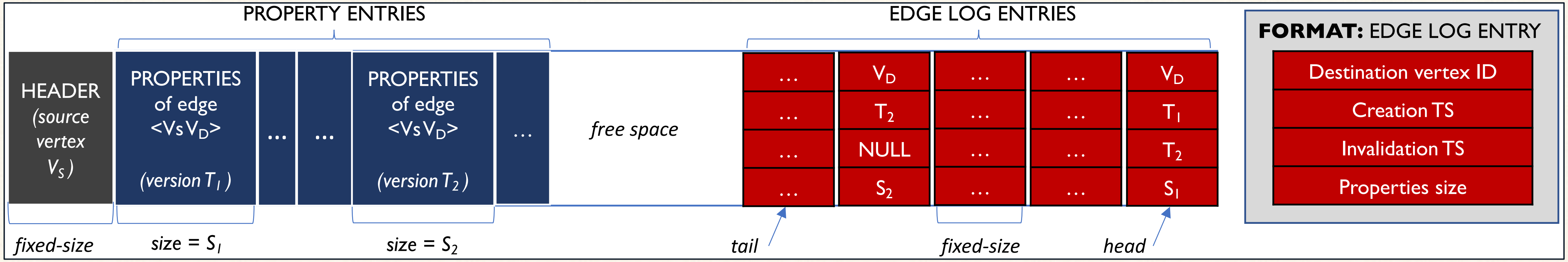


BENEFITS OF SEQUENTIAL SCANS

- Better locality
  - Cache utilization
- Sequential execution flow
  - Leverages CPU pipelining and prefetching
  - Reduces the likelihood of branch mispredictions
- Huge gap between pointer-based and sequential data structures
- Total latency improvement
  - 20× over LSMT
  - 18× over linked list
  - 4.5× over B+ tree.

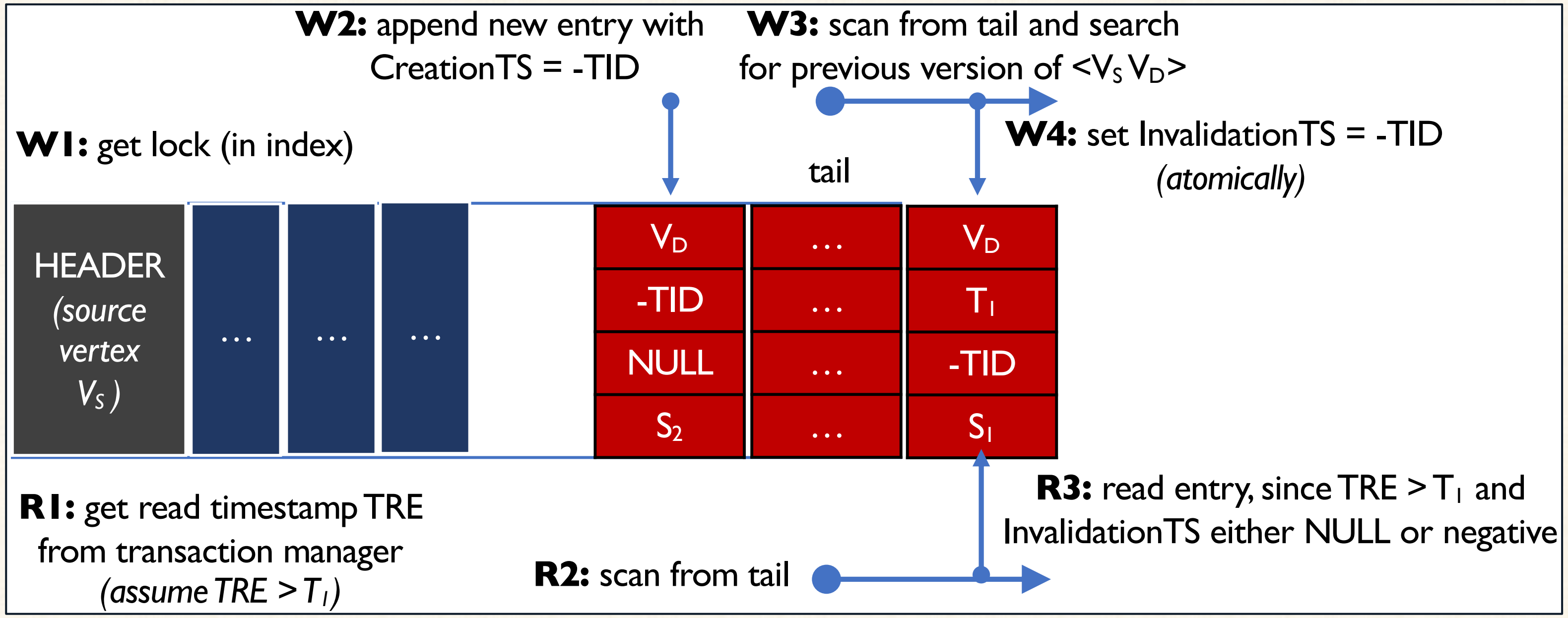
TRANSACTIONAL EDGE LOG

- Fixed-size "dynamic" array
  - Adapts to skew
- Append-only log
  - O(1) insertion
  - Multi-versioning: snapshots and temporal analytics



TRANSACTIONS + SEQUENTIAL SCANS

- Reads do not need locks
- Writes: double timestamps
  - Atomic timestamp access
  - 64-bit cache-aligned words



TRANSACTIONAL WORKLOAD

- LinkBench benchmark
  - Facebook's back-end graph storage workload
  - RocksDB: Facebook's back-end storage
- LiveGraph is a good match for latency-sensitive workloads
  - Sub-millisecond tail latency

Storage	Optane SSD			NAND SSD		
System	LiveGraph	RocksDB	LMDB	LiveGraph	RocksDB	LMDB
mean	**0.0450**	**0.1278**	1.6735	0.0915	**0.1804**	1.7495
P99	**0.2598**	**0.6423**	35.041	0.5995	**0.9518**	36.783
P999	**0.9800**	**3.5190**	74.610	1.2558	**4.0214**	77.906

Latency (ms)

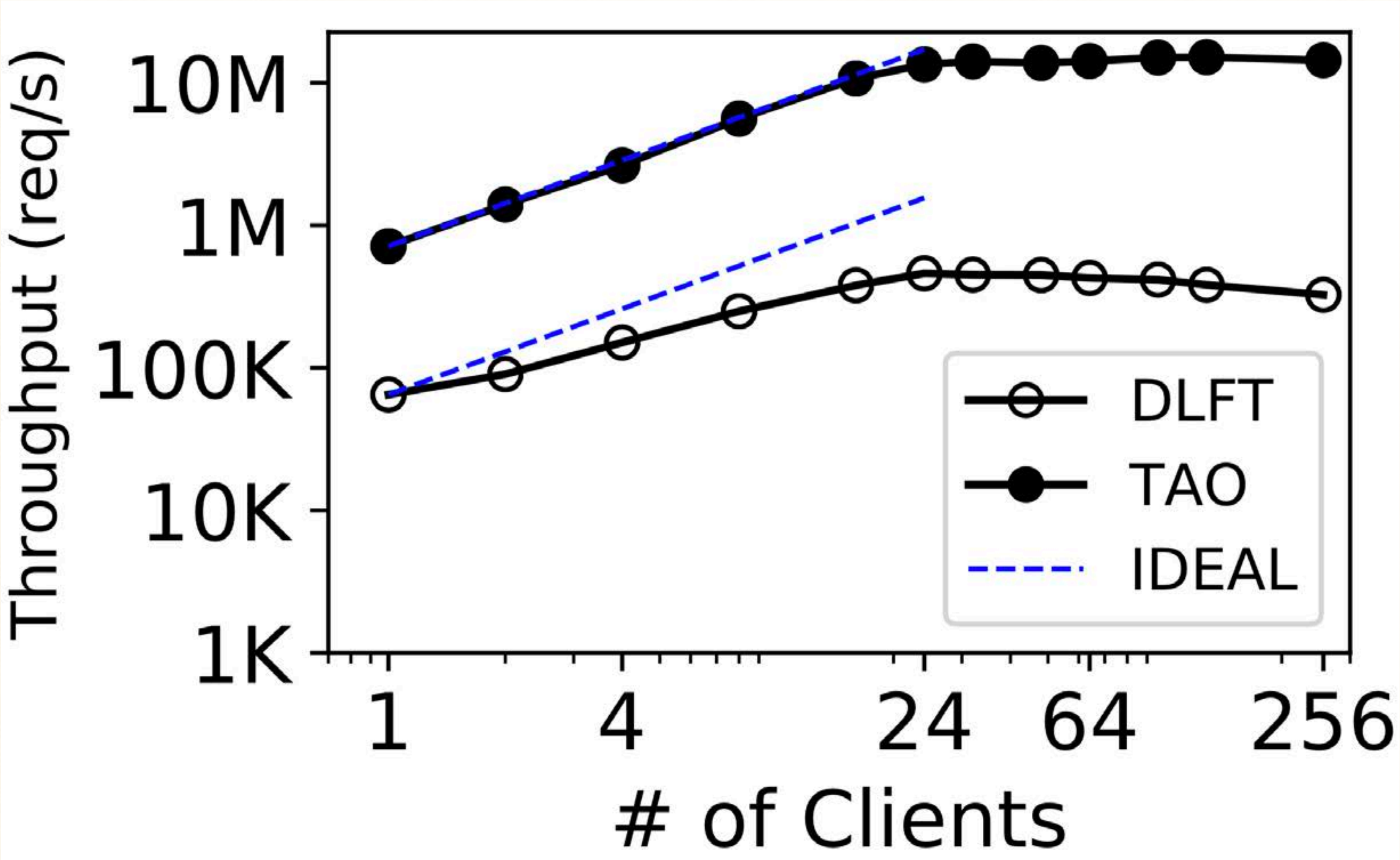
FRONT-END WORKLOAD

- Nano-second latencies!

Storage	Optane SSD			NAND SSD		
System	LiveGraph	RocksDB	LMDB	LiveGraph	RocksDB	LMDB
mean	**0.0039**	0.0328	**0.0109**	**0.0041**	0.0330	**0.0110**
P99	**0.0065**	0.0553	**0.0162**	**0.0066**	0.0581	**0.0162**
P999	**0.6763**	4.8716	**2.0703**	**0.6510**	4.8776	**2.1120**

Latency (ms)

SCALABILITY



REAL-TIME ANALYTICS

- LDBC Social Network Benchmark (SNB), in-memory
  - Short reads, transactional updates (possibly involving multiple objects)
  - Complex reads: multi-hop traversals and analytical processing including filters, aggregations, and joins

System	LiveGraph	Virtuoso	PostgreSQL	TigerGraph
Complex-Only	9,106	292	3.79	185
Overall	9,420	259	52.4	–

Throughput (ops/s)

717 without Query 14

TRUE REAL-TIME

- Interactive/web analytics must be in the millisecond range!

System	LiveGraph	Virtuoso	PostgreSQL
Complex read 1	7.00	23,101	371
Complex read 13	0.53	2.47	10,419
Short read 2	0.22	3.11	3.31
Updates	0.37	0.93	2.19

Average request latency (ms)

VERTEX-CENTRIC COMPUTATION

- Comparison between
  - Running in-database computation with LiveGraph
  - Export to Gemini, dedicated system using compressed read-only storage (CSR)
- Longer running time but no data export delay

System	LiveGraph	Gemini
ETL	-	1520
PageRank	266	156
ConnComp	254	62.6

Running time (ms)

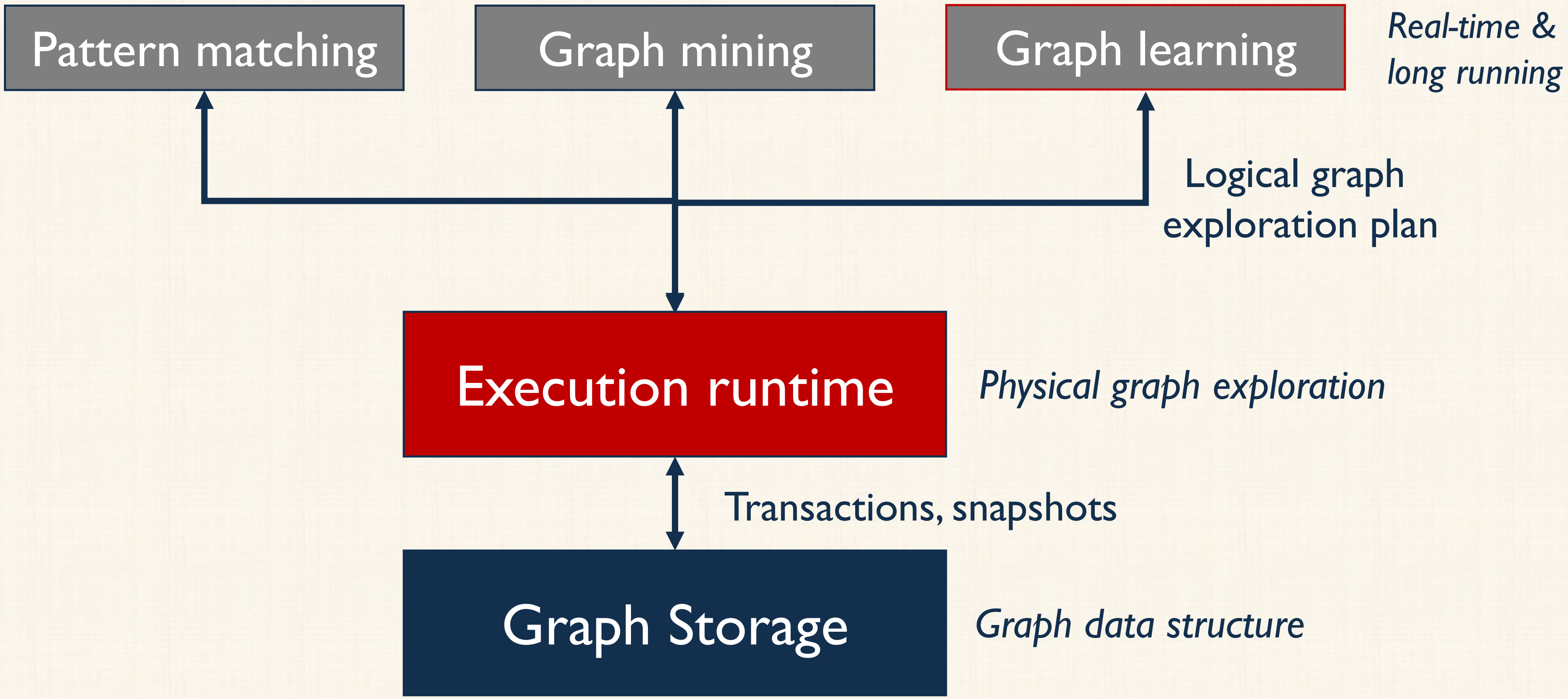
FUTURE WORK

- Scale out to distributed system
- Multi-hop locality/partitioning
- Improved property indexing

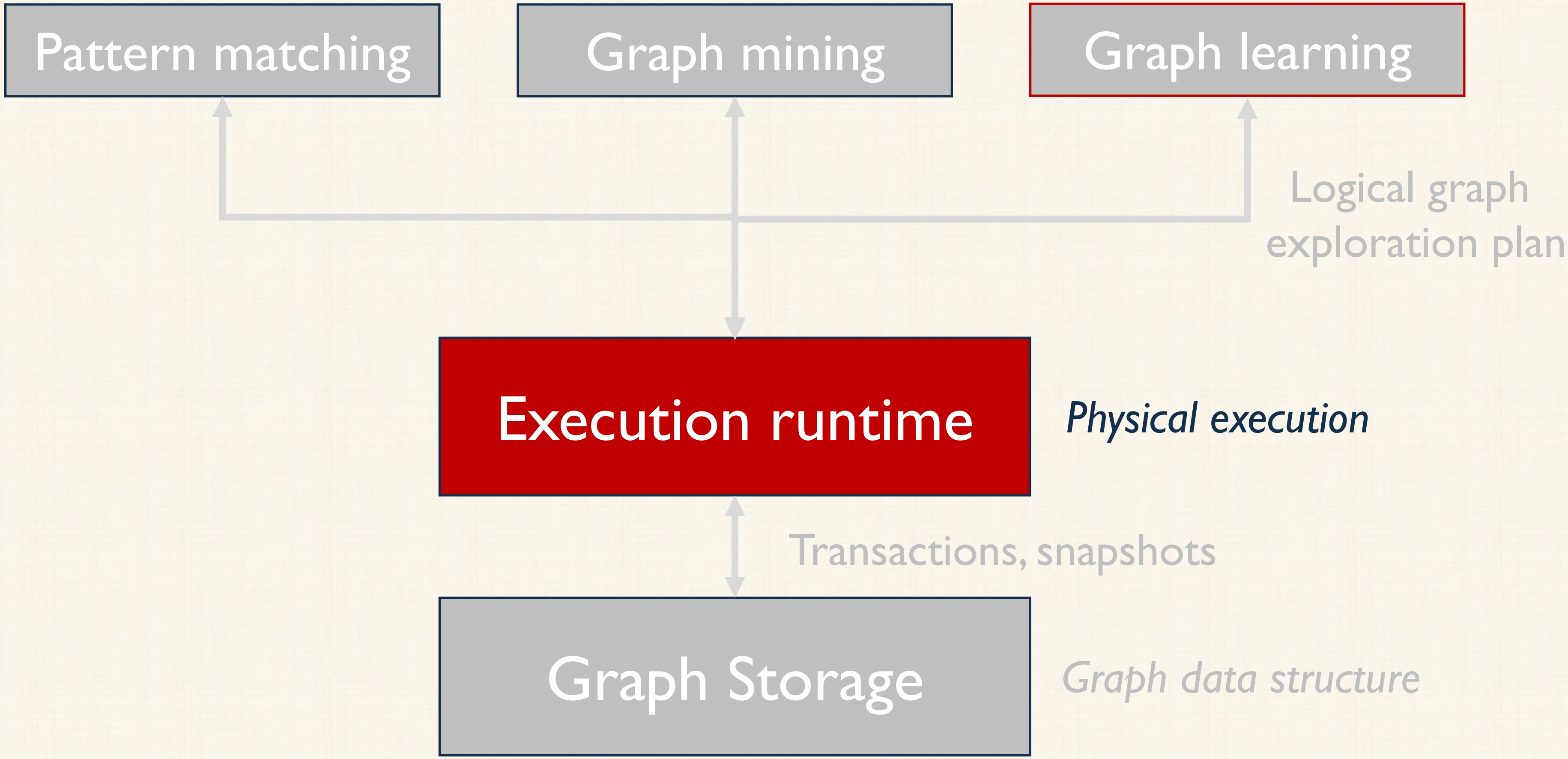
QUERYING CONNECTED DATA

CPU-EFFICIENT PHYSICAL EXECUTION

VISION

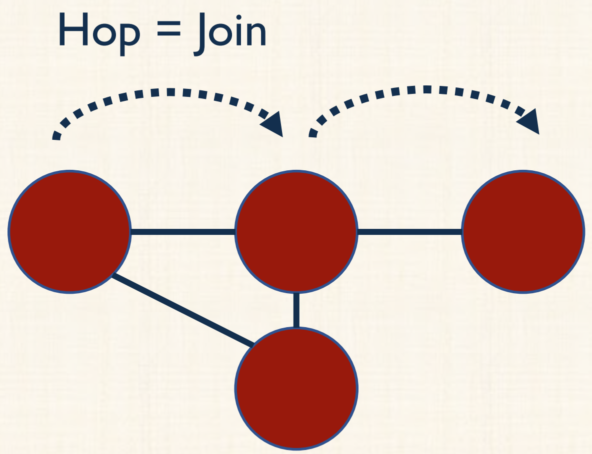

Pattern matching
Graph mining
Graph learning
Real-time &
long running
Logical graph
exploration plan
Execution runtime
Physical graph exploration
Transactions, snapshots
Graph Storage
Graph data structure

VISION


Pattern matching
Graph mining
Graph learning
Logical graph exploration plan
Execution runtime
Physical execution
Transactions, snapshots
Graph Storage
Graph data structure

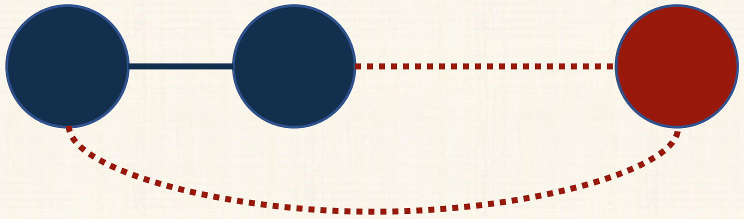
GRAPH PATTERN QUERIES

- Each “hop” is a join in the edge table
- Many graph queries are multi-hop
- This makes query optimization hard
  - Cardinality estimation gets harder at every join
  - Skew: few vertices have very high degree
  - Large intermediate results (e.g. structural or point-to-point path queries)



WORST-CASE OPTIMALITY (WCO)

- WCO: query complexity is the same as the size of the results
  - Example: triangle query should have complexity $O(|E|^{3/2})$
- Multi-way joins
  - Extend partial match by one vertex (not edge) at a time
  - Perform two joins at once
- **Set intersection**

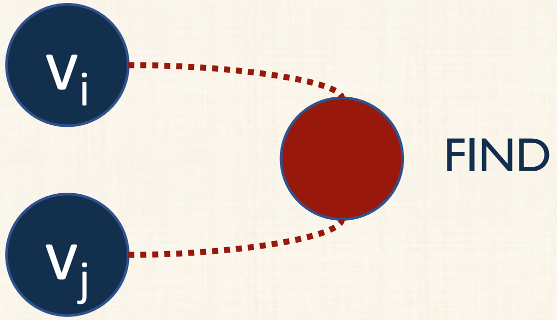


Multi-way join
(2 edges at once)

SET INTERSECTION BOTTLENECK

- Set intersection dominates running time
  - Frequent comparisons → frequent branch mispredictions
  - Need to fetch lots of data to cache → poor caching

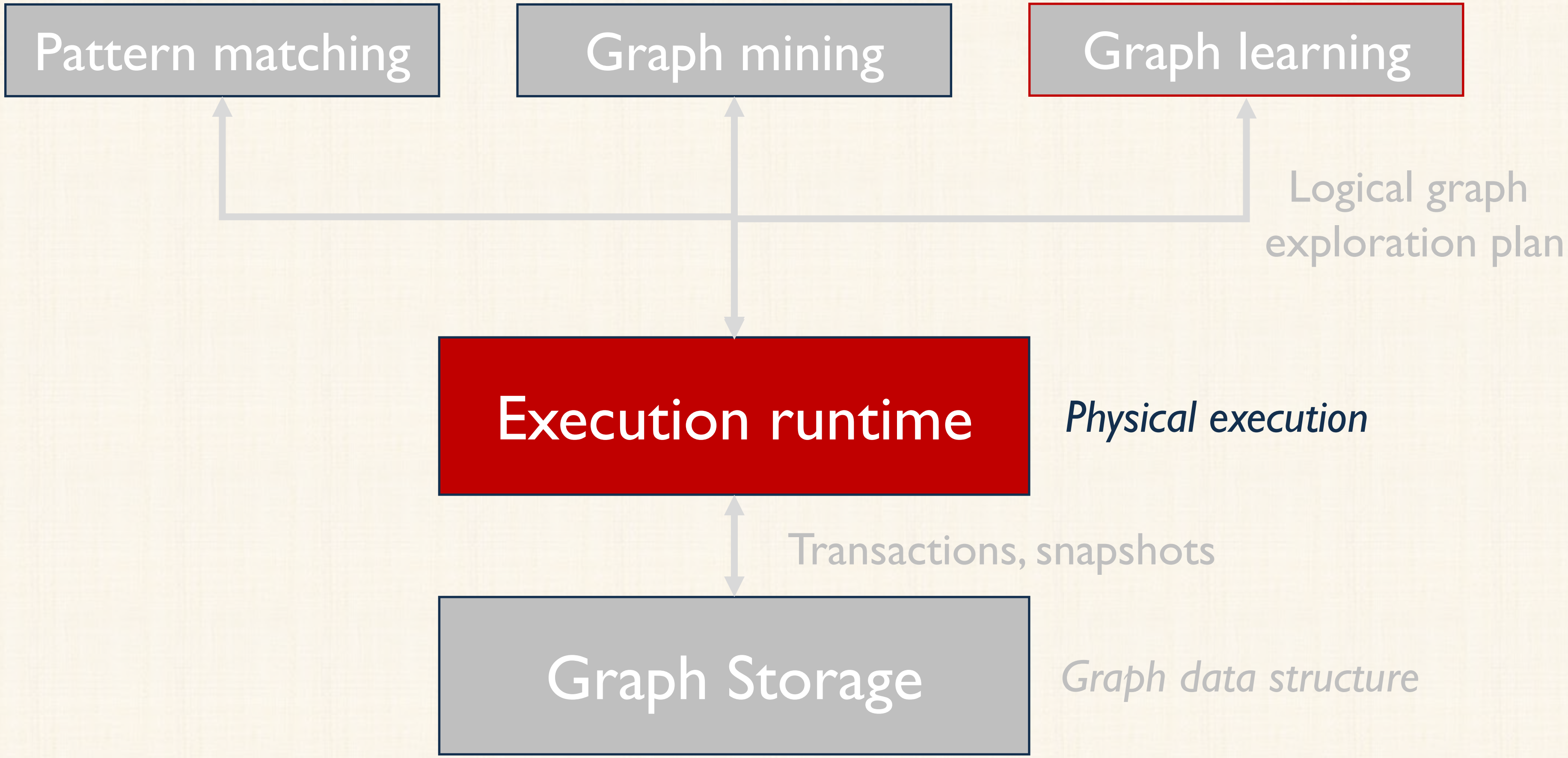
v_i	1	13	32	143	…			
v_j	4	5	13	43	143	178	251	…



VECTORIZER

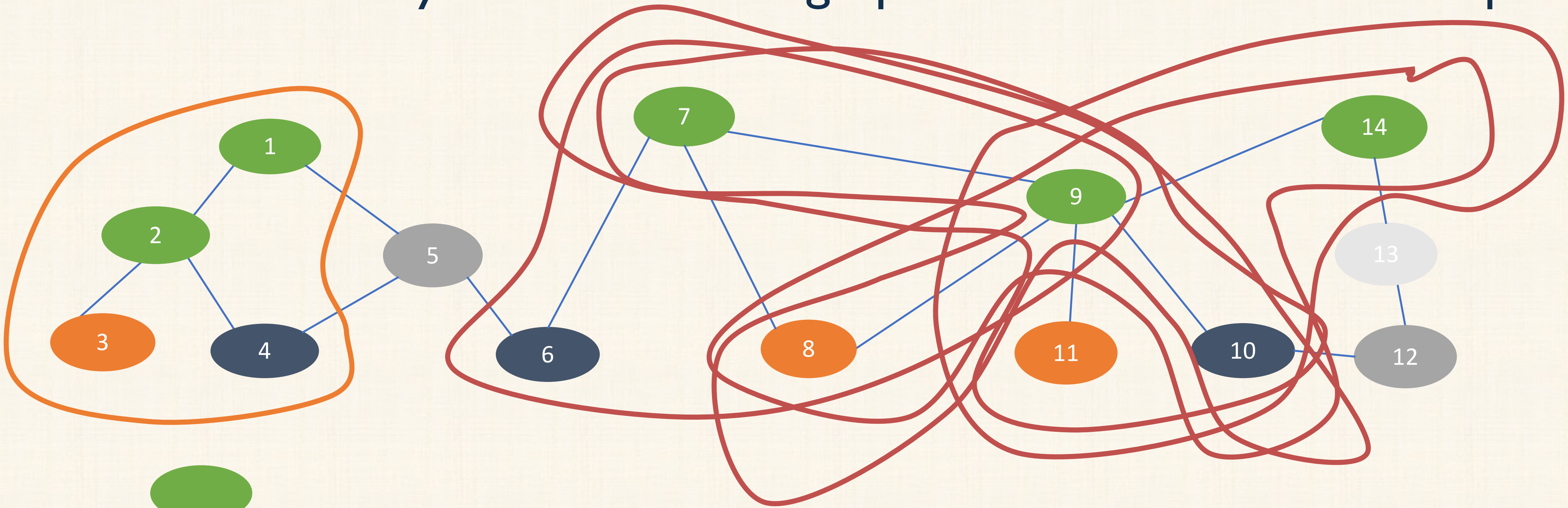
- Goal: optimize CPU efficiency
  - Cache efficiency: Data compression
  - Avoid branch mispredictions
  - SIMD operations
- Dynamic data: Cannot afford expensive pre-processing
- Vectorizer: On-the-fly vectorization
  - SIMD friendly data structures
  - Materialization and reuse of these data structures
  - **> 3x speedup** compared to state of the art graph tools
  - **> 10x speedup** compared to RDBMS

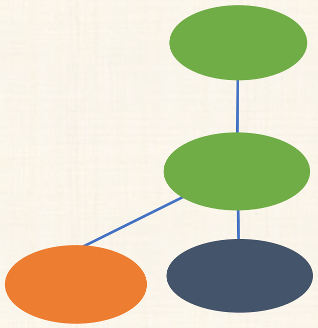
BEYOND GRAPH QUERIES?



FREQUENT SUBGRAPH MINING

- Search for initially **unknown** subgraphs that turn out to be frequent

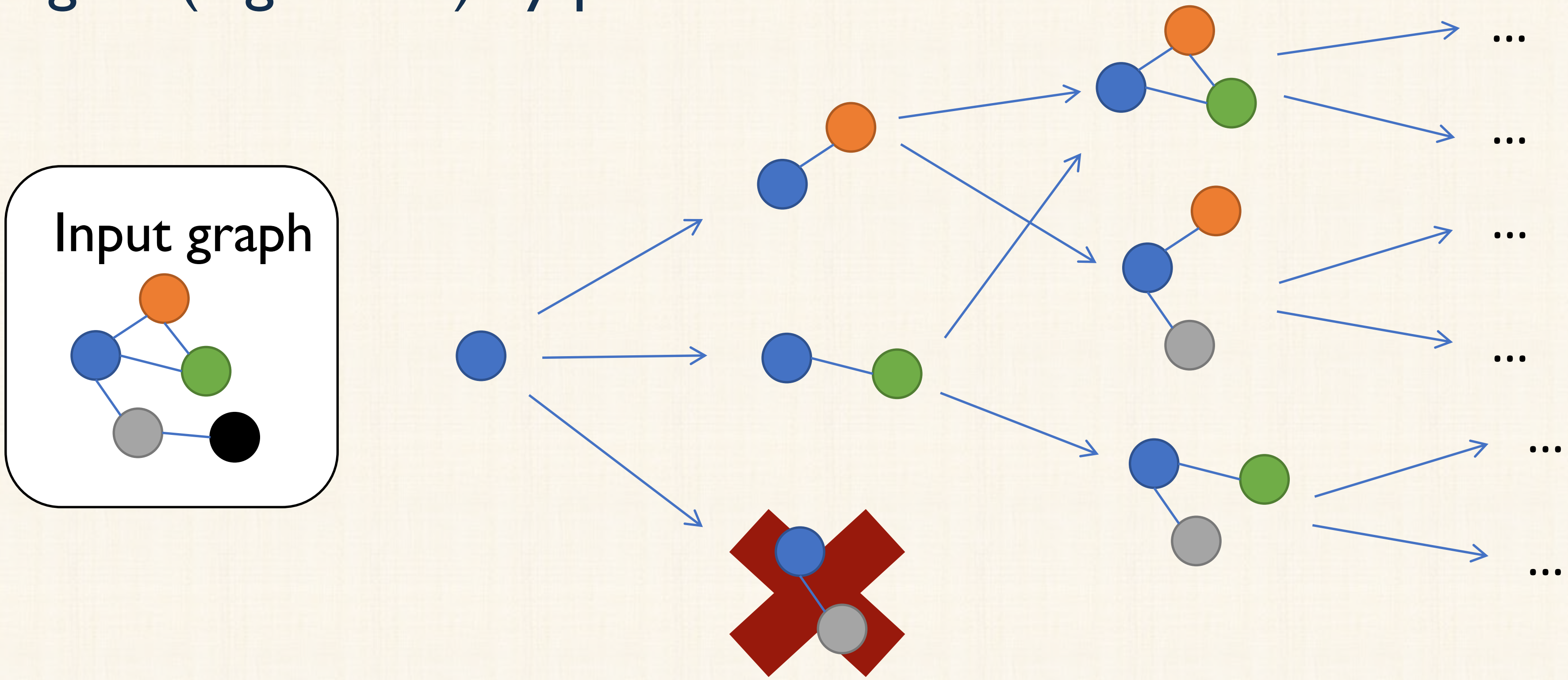




← Is a Frequent Subgraph

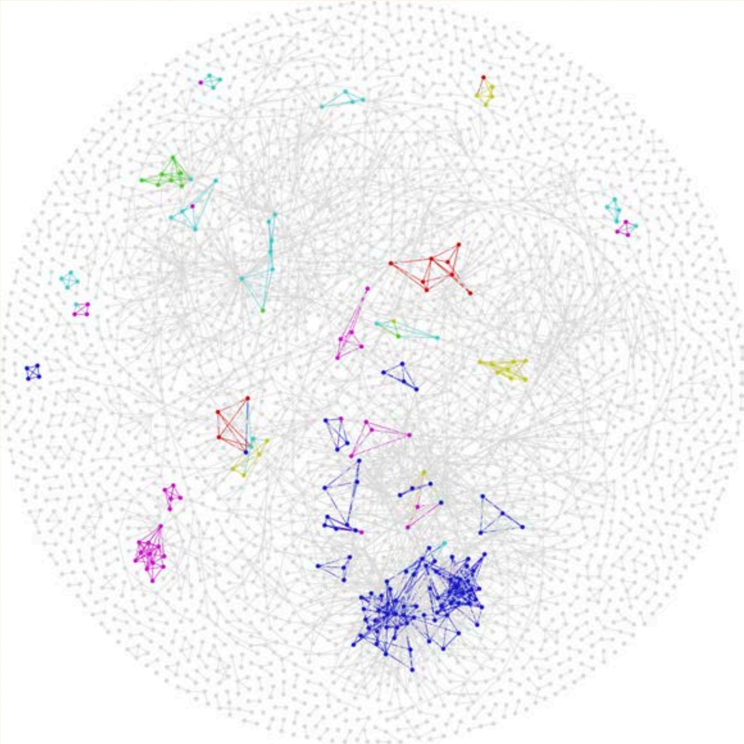
GRAPH EXPLORATION PROCESS

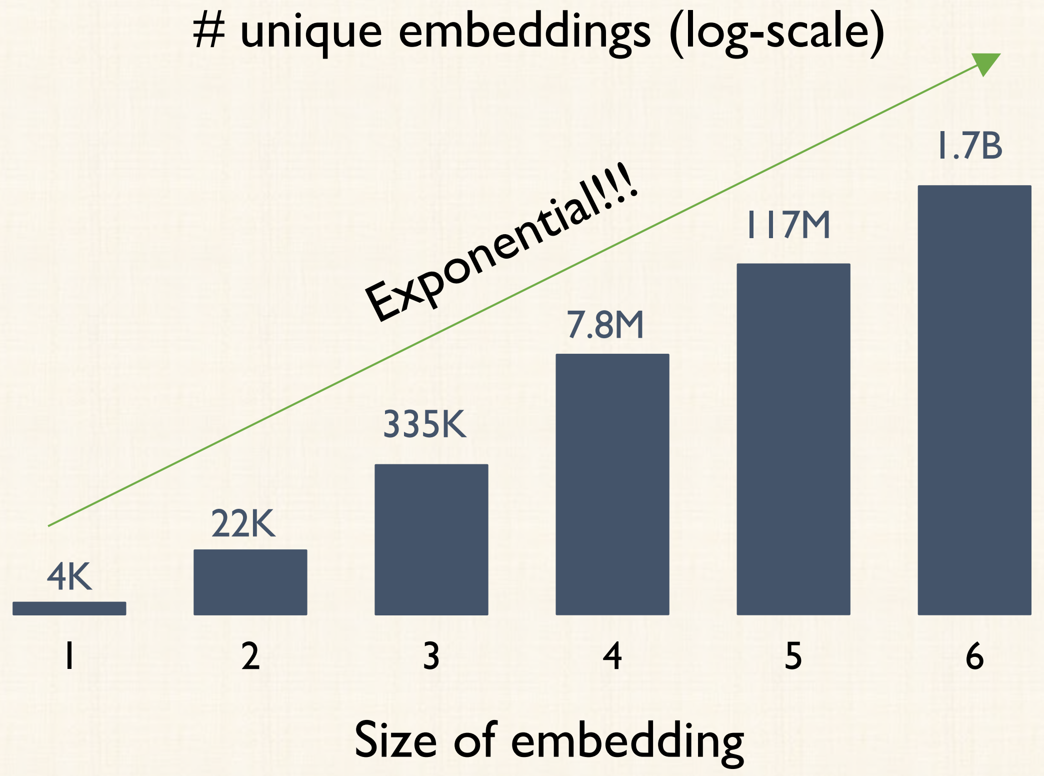
- Enumerate (& prune) embeddings
- Aggregate (e.g. count) by pattern



CHALLENGES

- Exponential number of embeddings





ARABESQUE

- New execution model & system
  - Think Like an Embedding
  - Purpose-built for distributed graph mining
  - Hadoop-based
- Contributions
  - Simple & Generic API
  - High performance
  - Distributed & Scalable by design

API EXAMPLE: CLIQUE FINDING

```
boolean filter(Embedding e) {
    return isClique(e);
}
void process(Embedding e) {
    output(e);
}
boolean shouldExpand(Embedding embedding) {
    return embedding.getNumVertices() < maxsize;
}
boolean isClique(Embedding e) {
    return e.getNumEdgesAddedWithExpansion()==e.getNumberOfVertices()-1;
}
```

Previous

state of the art

(Mace, centralized)

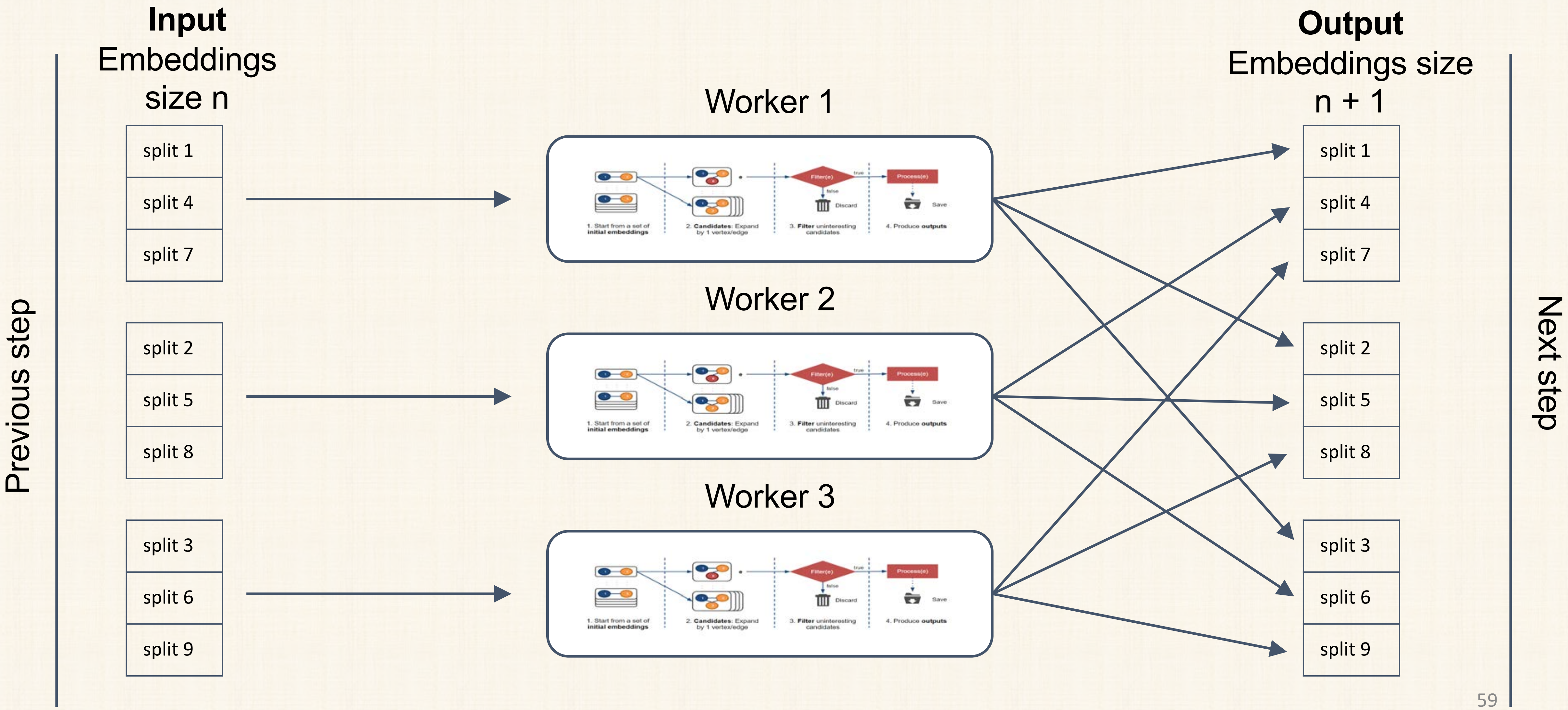
4,621 LOC

FREQUENT SUBGRAPH MINING

- First distributed implementation
- 280 lines of java code
  - … Of which 212 compute frequency metric

- Baseline (Grami): 5,443 lines of Java code

ARABESQUE ARCHITECTURE



KEY FUNCTIONALITIES

- Avoiding redundant work
- Compression and management of huge intermediate state
- Load balancing
- Efficient pattern aggregation

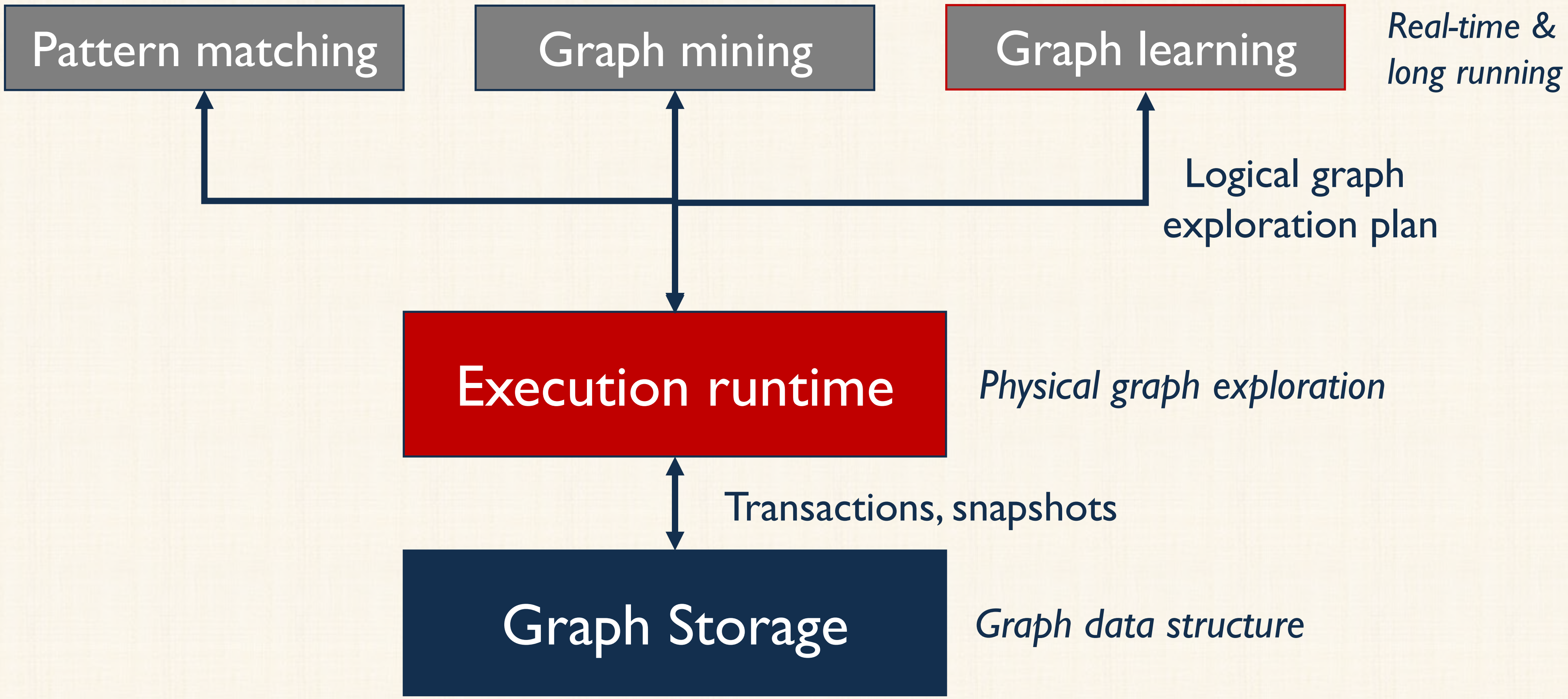
LIMITATIONS OF API

- Limited control over exploration
  - Not ideal when looking for a specific pattern
- No support for sampling/random traversals
- Related APIs
  - NScale, G-Miner, ASAP, Fractal, …
- Finding the right API is still an active research topic

PARALLEL GRAPH EXPLORATION

- Can we leverage parallel hardware like GPUs?
- Example: graph learning
  - Training uses standard GPU tools for neural networks
  - But mining graph features on GPUs is an open problem
- Challenges
  - Limited CPU-GPU bandwidth
  - Scalability to large graphs
  - Random access and skew make SIMD operations ineffective

VISION



Pattern matching
Graph mining
Graph learning
Real-time &
long running
Logical graph
exploration plan
Execution runtime
Physical graph exploration
Transactions, snapshots
Graph Storage
Graph data structure

PICK THREE?

- Fresh results on dynamic data
- Complex data exploration
  - Random access
  - Query optimization hard
- Low-latency results

THANK YOU

MARCO SERAFINI
University of Massachusetts Amherst
marco@cs.umass.edu