Data Streams

Immanuel Trummer
itrummer@cornell.edu
www.itrummer.org

Outlook:
Beyond Relational Data

- Graph data
- Data streams
- Spatial data

Outlook:
Beyond Relational Data

- Graph data
- **Data streams**
- Spatial data

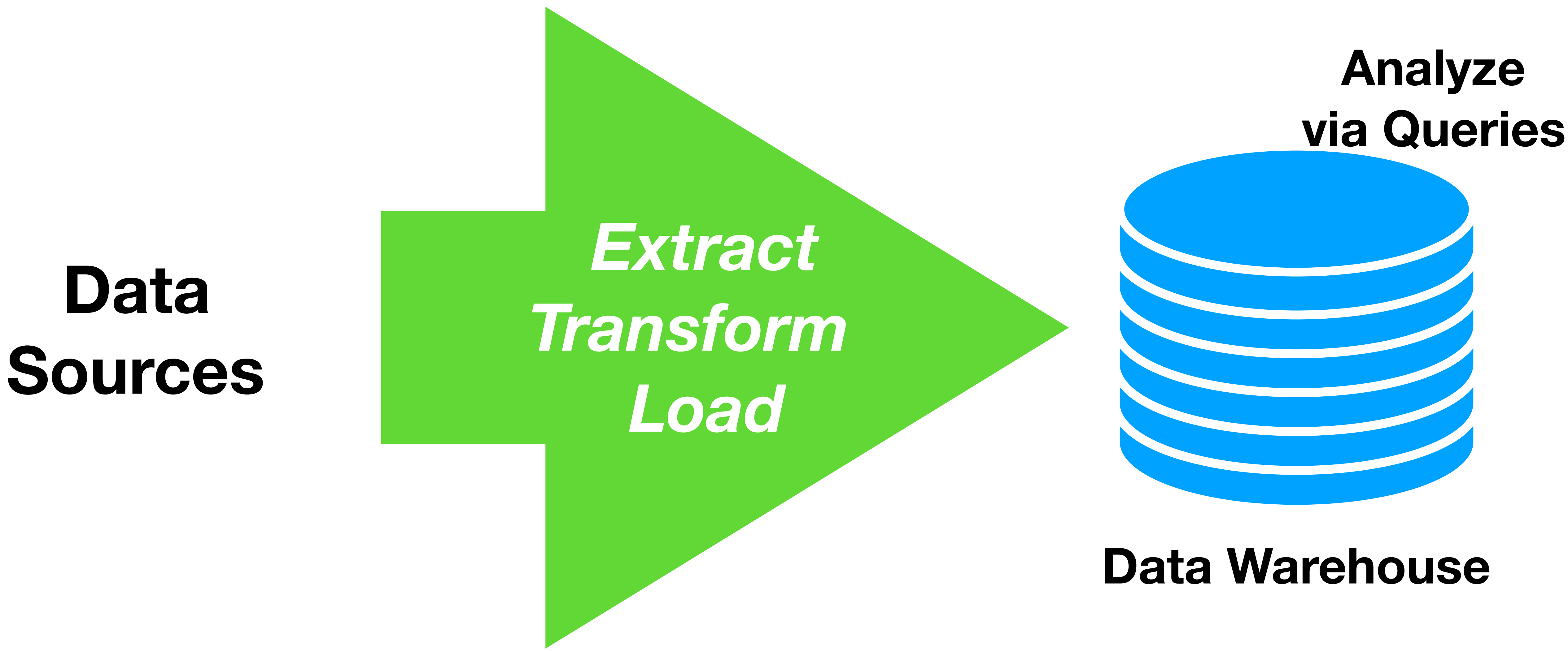
Reading List

- "*STREAM: the Stanford data stream management system*", 2003, Arasu et al.
- "*Streams and tables: two sides of the same coin*", 2016, Sax et al.
- ksqlDB Web site: https://ksqldb.io/
- "*LSM-based storage techniques: a survey*", 2019, Luo et Carey.

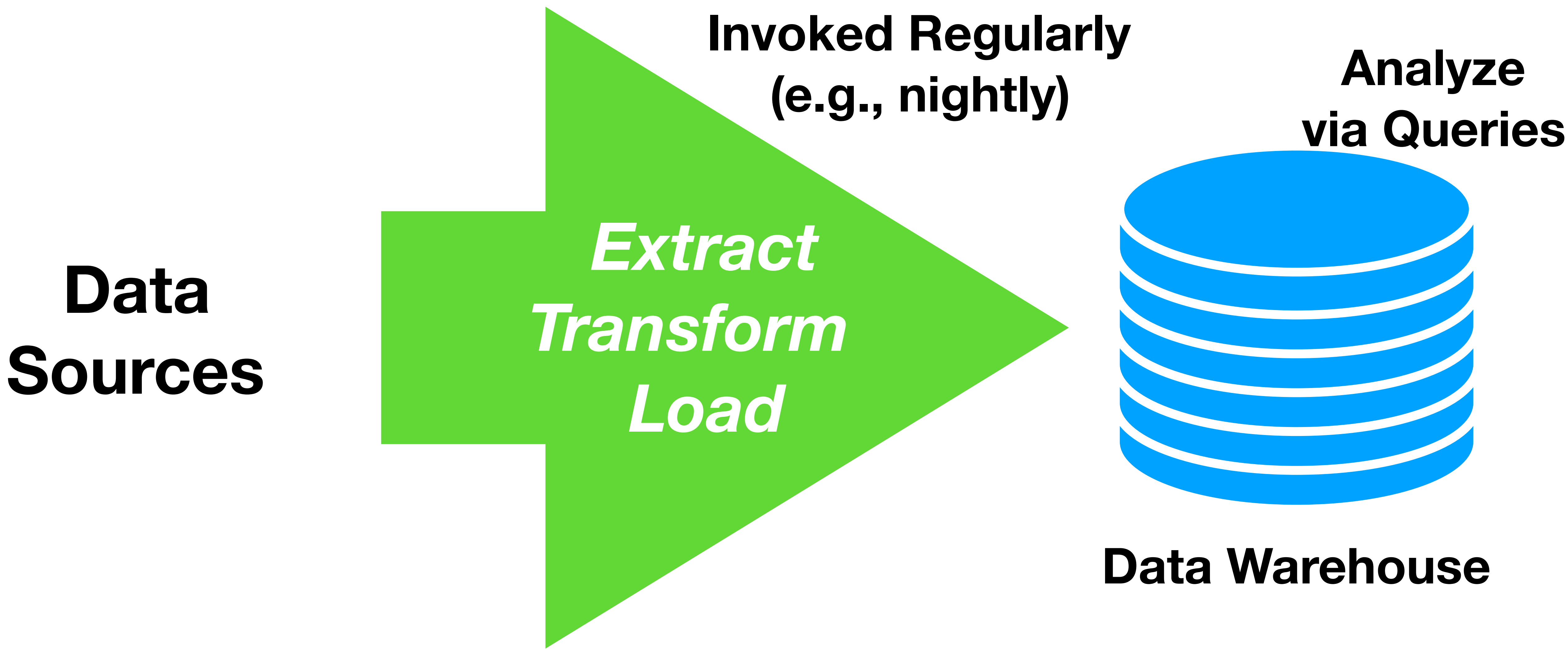
Data Streams

- Data is constantly being **generated**!
  - Stock market ticker
  - Network monitoring
  - Sensors ...
- May need to **react** to specific patterns in real time!
  - Fraud detection, medical intervention, stock sales, ...

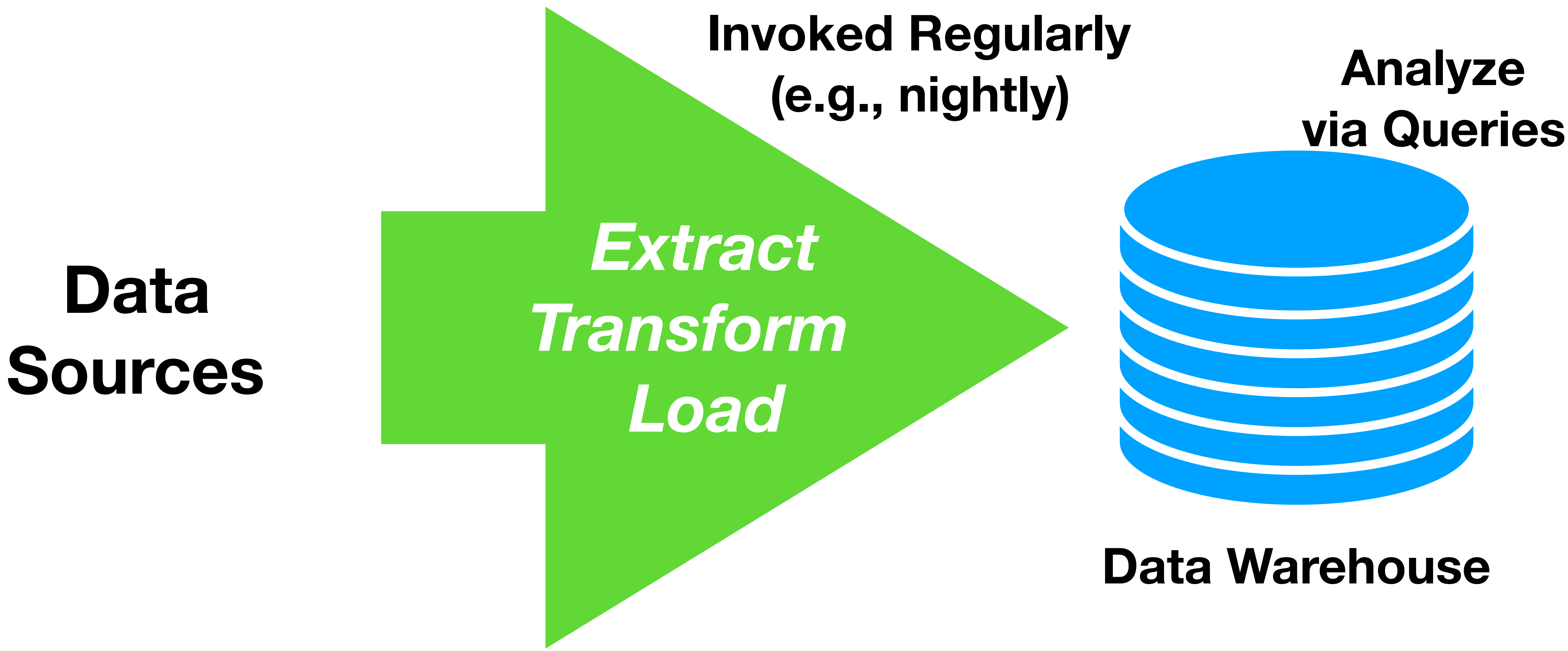
Traditional Data Ingestion



Traditional Data Ingestion



Traditional Data Ingestion



Unsuitable for Reacting in Real Time!

Stream Data Requirements

- Traditional **ETL** supports queries on static snapshots
- **Delay** between snapshots is often too high
- Streams keep **generating** new data with high frequency
- Query results keep **changing** (for query on stream)
- Hence, it is useful to keep queries **running**

Stream Data Management

	Database Management	Stream Data Management
Data	**Static**	**Dynamic**
Queries	**Dynamic**	**Static**

Data Stream Topics

- STREAM System (~2003)
    - First "Stream Data Management System"
- ksqlDB (~2020)
    - Recent system for distributed stream processing

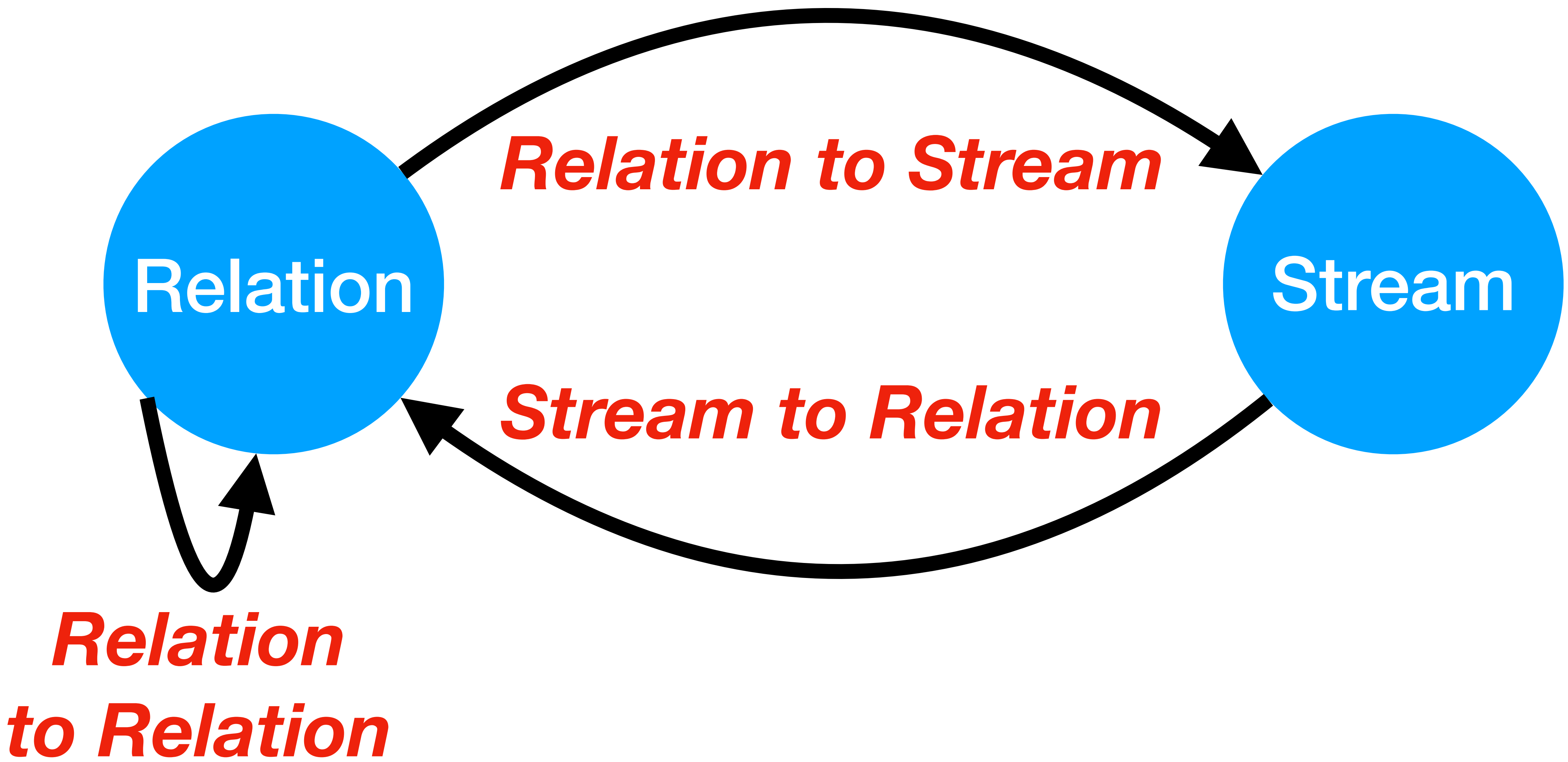
Data Stream Topics

- **STREAM System (~2003)**
    - First "Stream Data Management System"
- ksqlDB (~2020)
    - Recent system for distributed stream processing

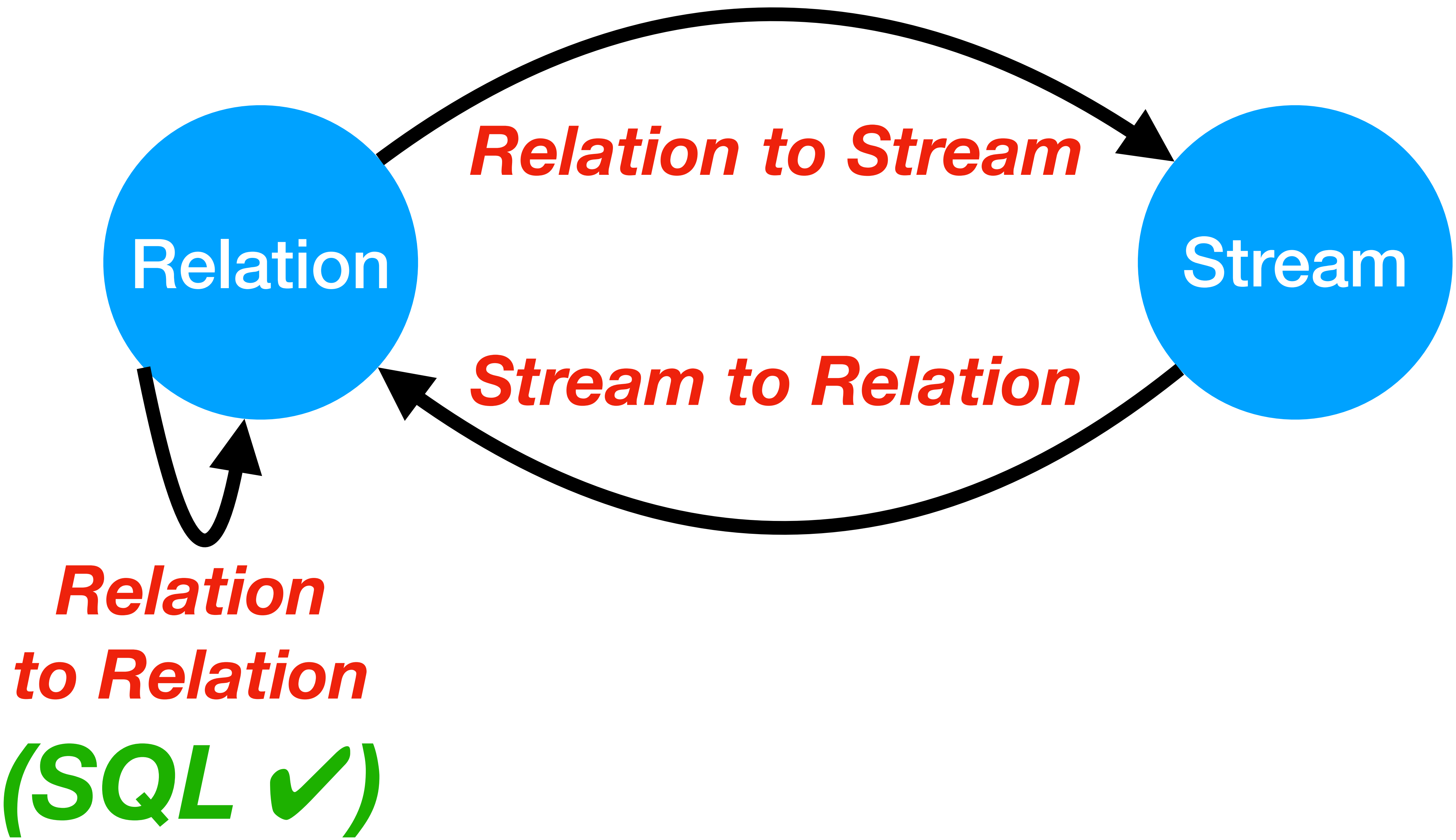
Data Types

Data**base** Management System	Data **Stream** Management System
Relation R: **static** (until changed explicitly)	Relation R(t): **varies** over time
	Stream S: timestamped tuples

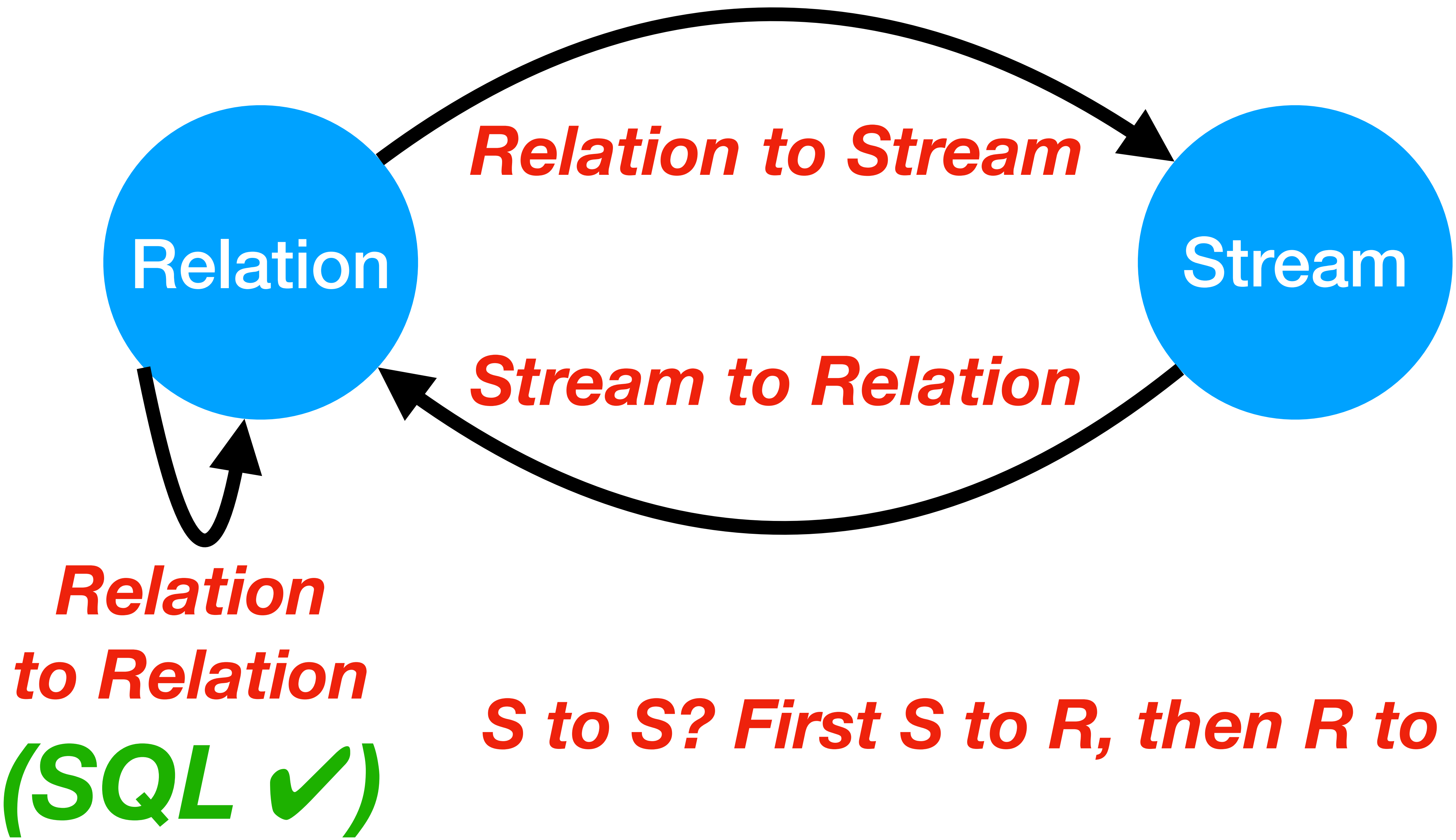
Classes of Operators



Classes of Operators



Classes of Operators



S to S? First S to R, then R to S.

Stream to Relation

- Relation R(t) is specified as a **window** over stream S
- Tuple-based sliding window: **S [Rows N]**
  - R(t) contains N tuples from S with highest timestamps
- Time-based sliding window: **S [Range T]**
  - R(t) contains tuples from S starting from Now() - T
- Partitioned sliding window: **S [Partition by A1, A2, ... Rows N]**
  - Separate windows for each value combination in A1, ...

Relation to Stream

- **Istream(R)**: R's inserted tuples with insertion timestamp
- **Dstream(R)**: R's deleted tuples with deletion timestamp
- **Rstream(R)**: R's current content with current timestamp

Example Queries

- SELECT Avg(price) FROM StockPriceStream [Rows 10]
  WHERE stock = 'AAPL'

- SELECT * FROM Customers C
    JOIN Orders [Range 2 Minutes] O
    ON (C.customerKey = O.customerKey)

- SELECT Istream() FROM (
    SELECT * FROM Clicks[Range 30 Seconds] C
    JOIN Advertisers A ON (C.advKey = A.advKey)
)

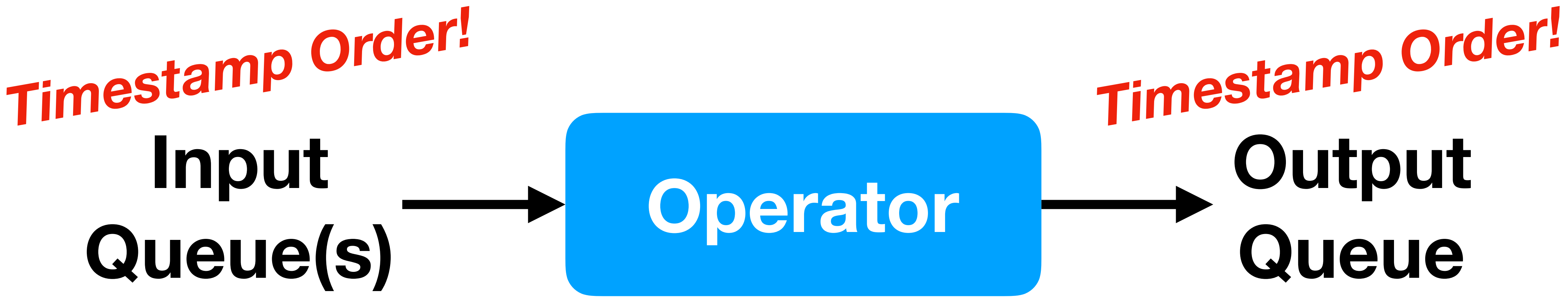
What is the semantics of those queries?

Query Processing

- Input query is compiled into continuous **query plan**
- Query plan is composed from **standard operators**
- Operators exchange tuple **additions** and **deletions**
  - **Streams** produce only tuple additions
  - **Relations** produce additions and deletions

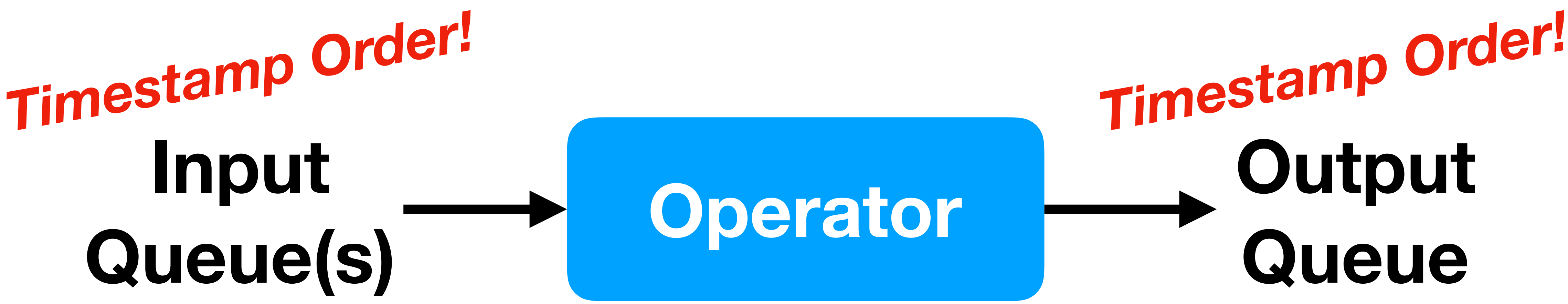
Operators

Operators



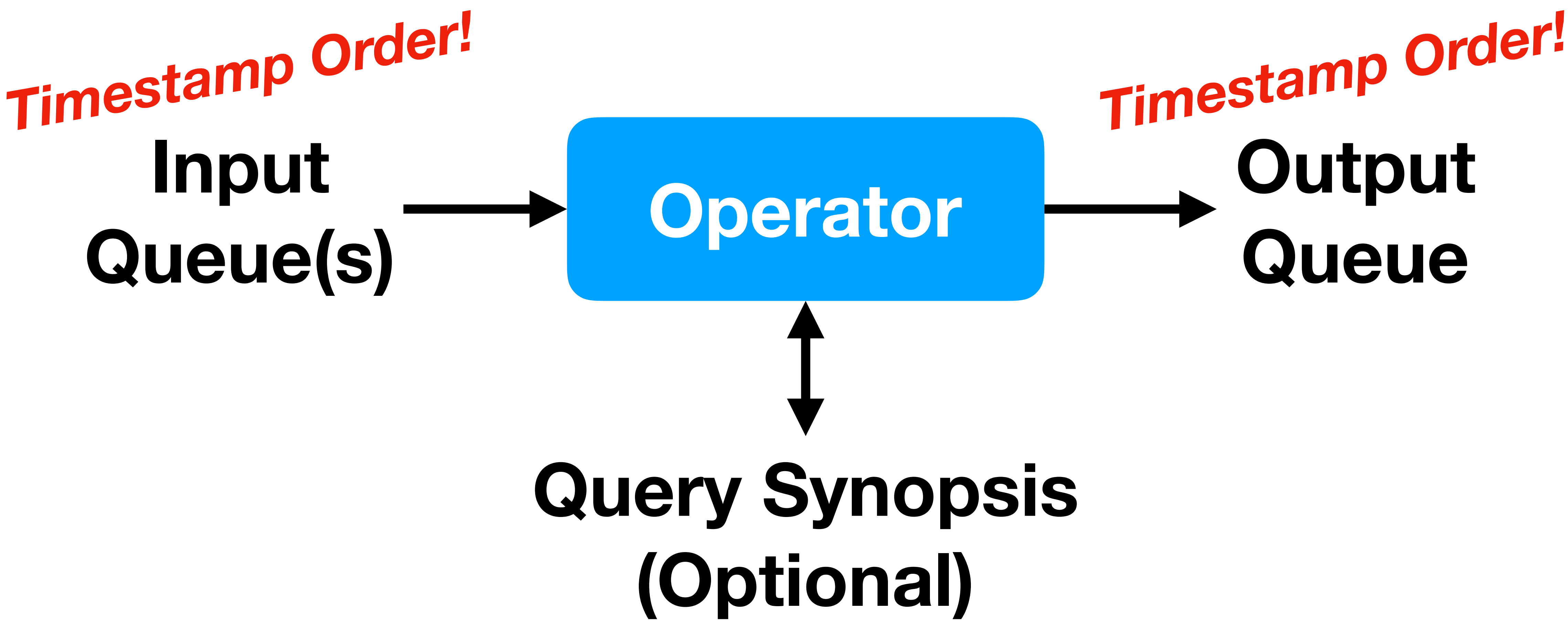
Timestamp Order!
Input
Queue(s)
Operator
Timestamp Order!
Output
Queue

Operators



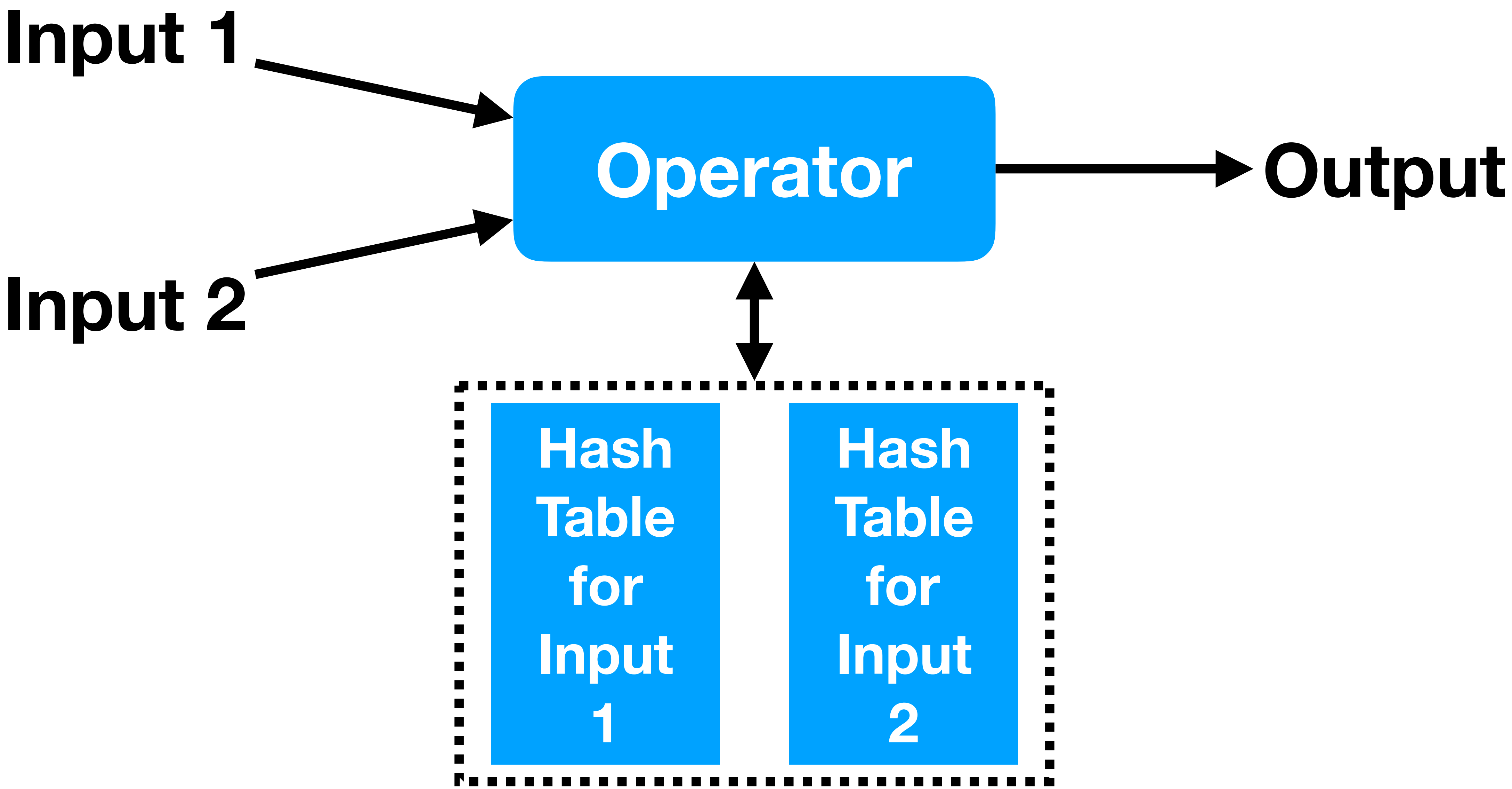
Enough for simple operators (e.g., filtering)

Operators



Others may store additional state in synopsis (e.g., hash table for join operators)

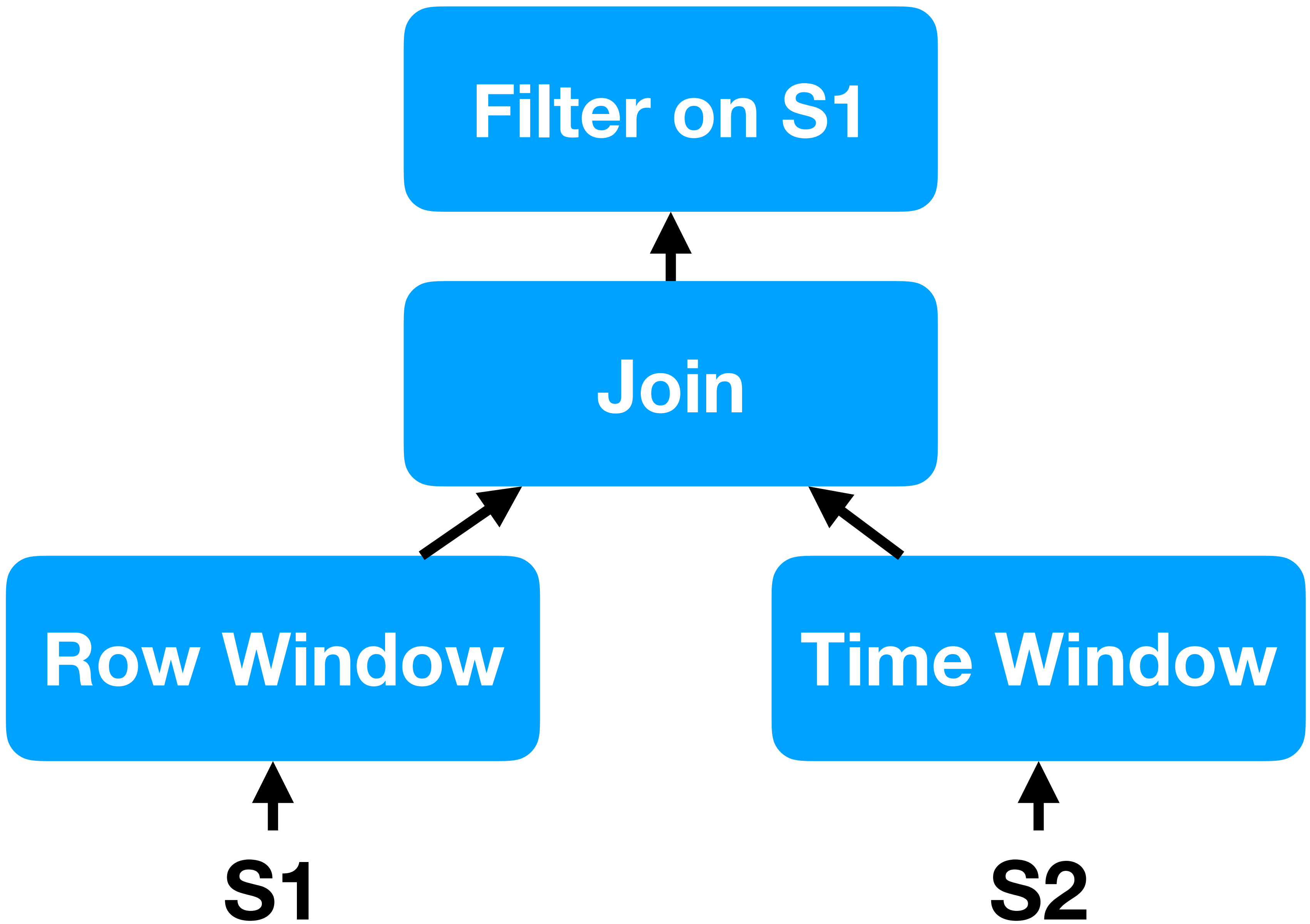
Join Operator



Join Algorithm

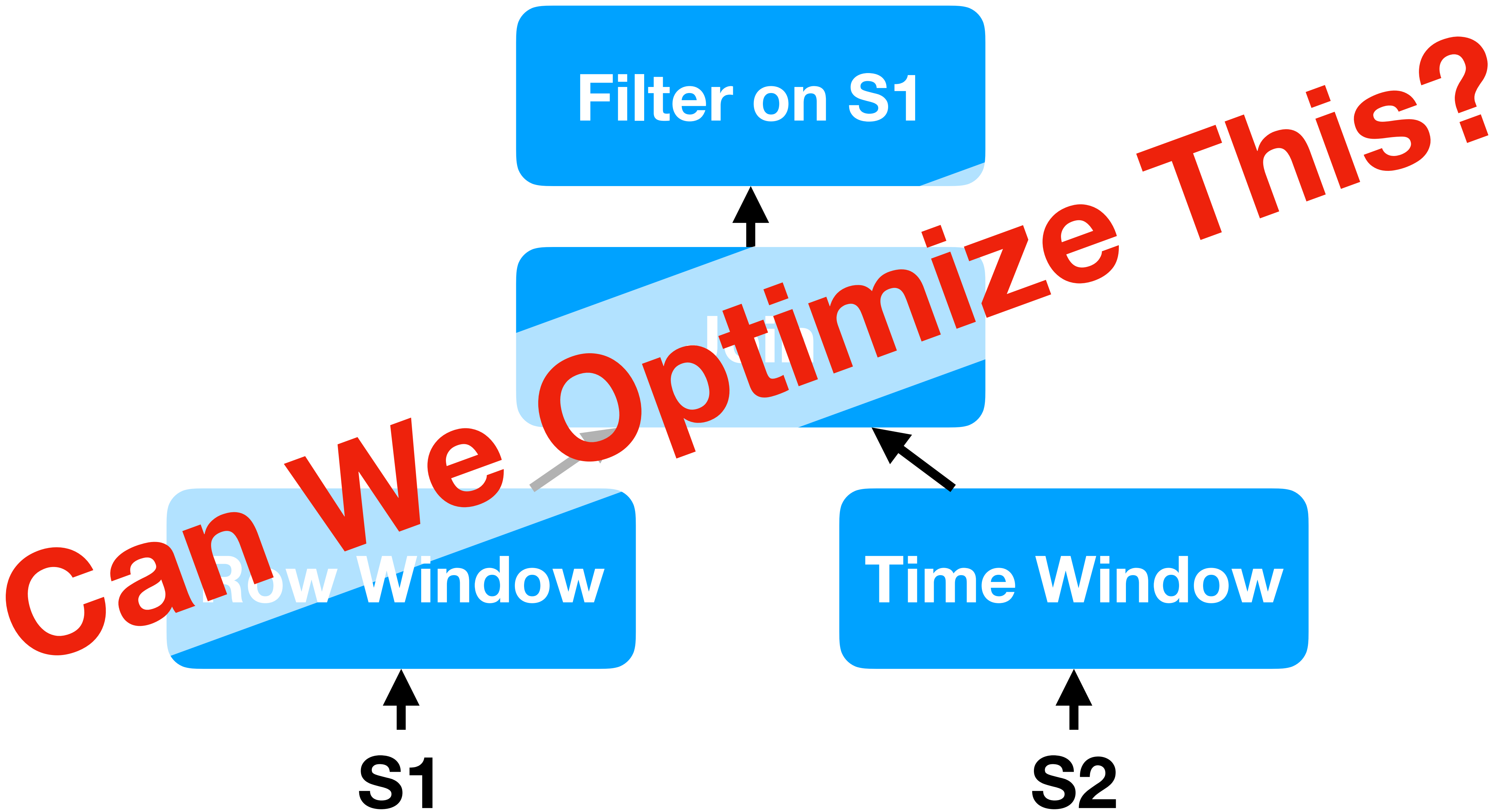
- Tuple **addition/deletion** in Input 1 Queue
  - Extract **join key** from added tuple
  - **Probe** hash table of Input 2 with key
  - Add/delete resulting join tuples to **output**
  - **Update** synopsis (hash table for Input 1)

Example Query Plan



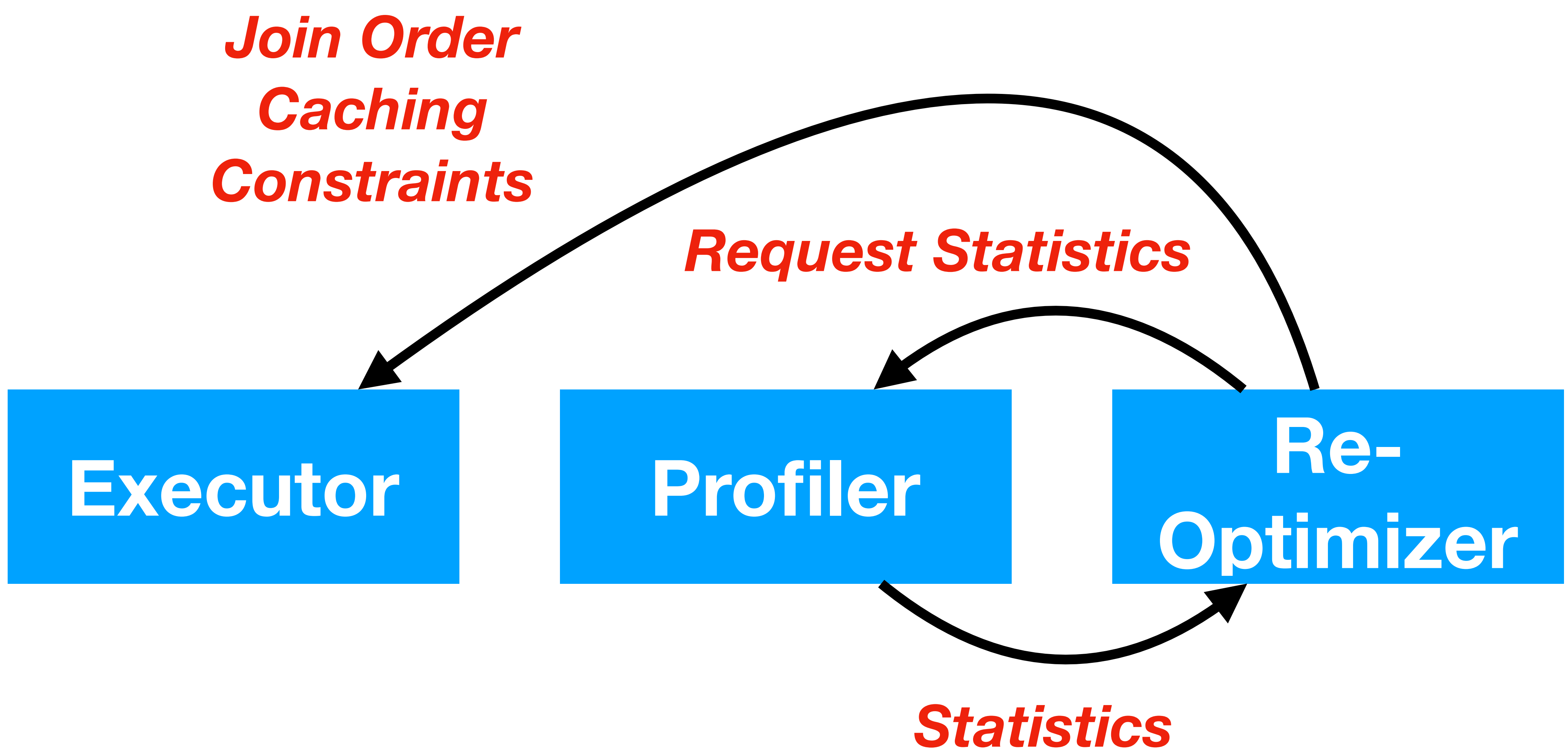
<u>Query</u>: SELECT * FROM S1 [Rows 1,000], S2 [Range 2 Minutes] WHERE S1.A = S2.A and S1.A > 10

Example Query Plan

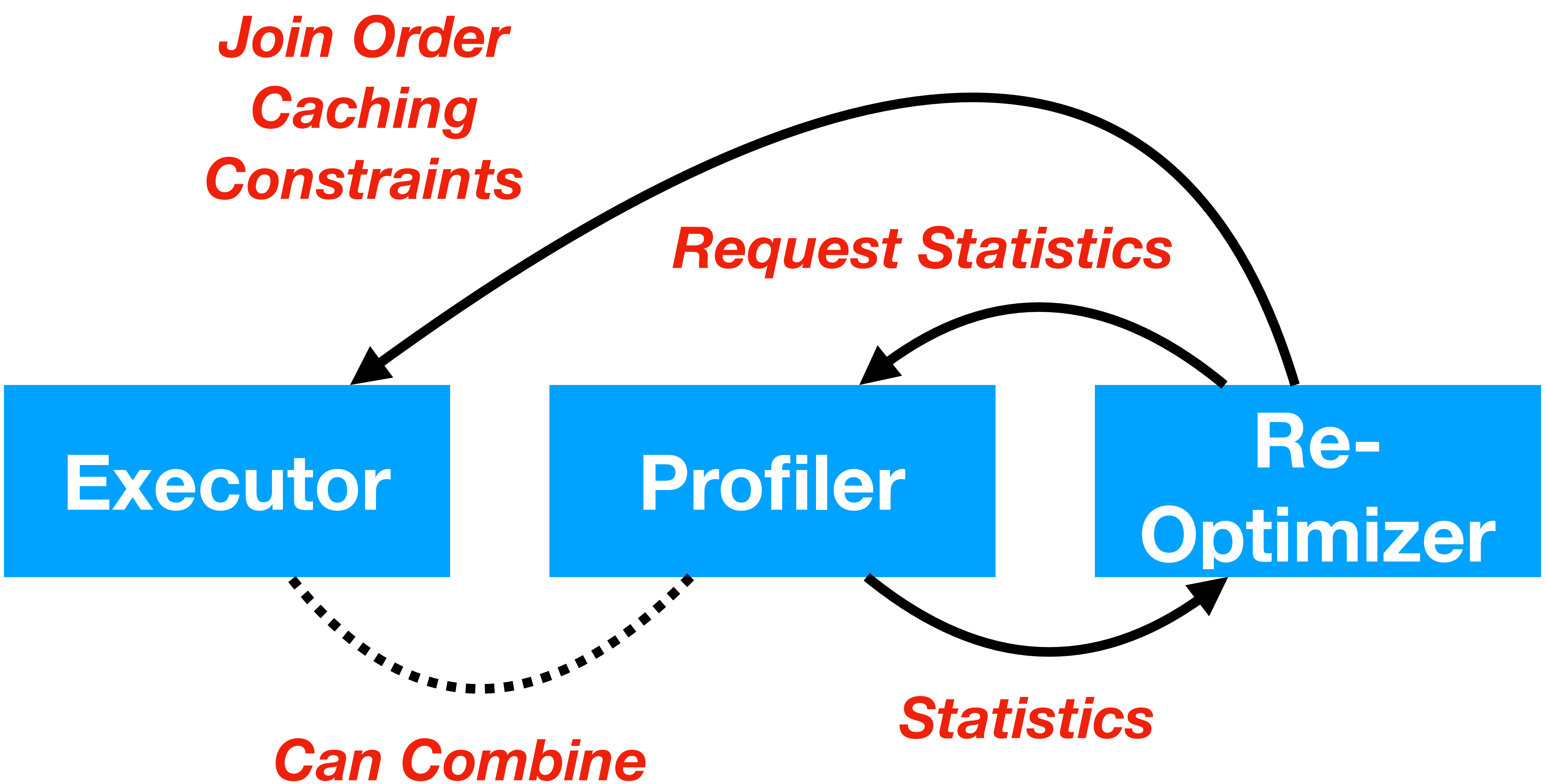


<u>Query</u>: SELECT * FROM S1 [Rows 1,000], S2 [Range 2 Minutes] WHERE S1.A = S2.A and S1.A > 10

Adaptive Query Planning



Adaptive Query Planning



Minimizing Space Requirements

- Very important for streams (**unbounded** size)
- Eliminate redundant data via **synopsis sharing**
- **Exploit constraints** to prune unnecessary data
- Shrink intermediate results via **optimized scheduling**

Synopsis Sharing

- Synopses of operators in same plan often **overlap**
- Storing synopses separately means **redundancy**
- Instead: **global** synopses with operator-specific views
- Can extend to merge synopses from **different plans**

Constraint Examples

- SELECT * FROM Orders [Rows Unbounded] O
  JOIN Fullfillment [Rows Unbounded] F
  ON (O.orderID = F.orderID)

- Requires **unbounded** synopses without constraints

- C1: Orders arrive before fullfillments - **what changes**?

- C2: Fullfillments clustered by orderID - **what changes**?

Constraint Types

- **Referential integrity** k-constraint
  - Refers to key-foreign key joins
  - Delay at most k between matching tuples arriving
- **Ordered-arrival** k-constraint
  - Stream elements at least k tuples apart are sorted
- **Clustered-arrival** k-constraint
  - Elements with same key can be at most k tuples apart

Can exploit each constraint for dropping tuples in certain scenarios

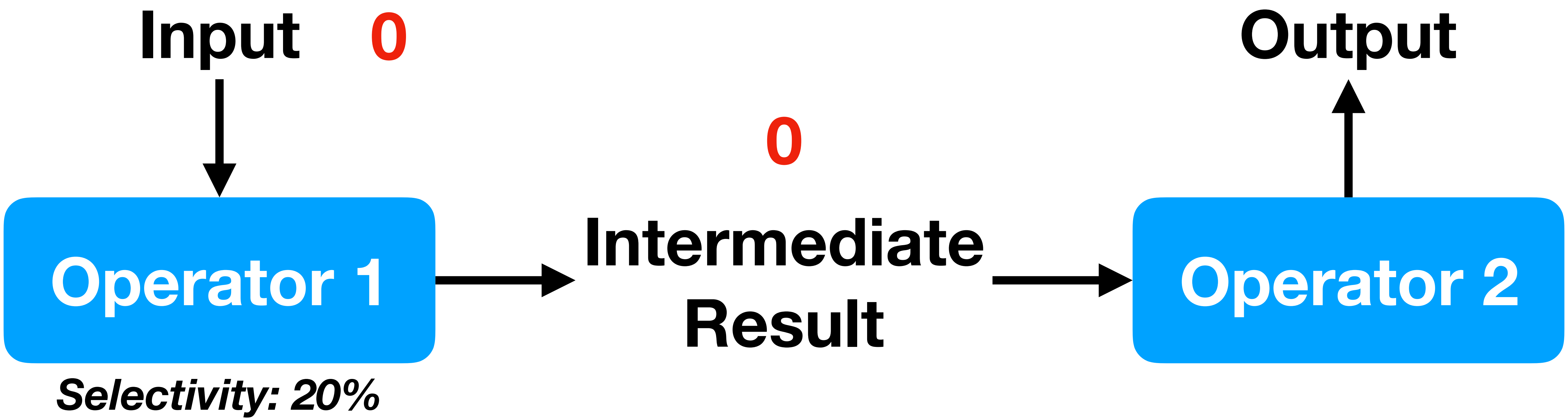
Scheduling Policies

- We have **flexibility** to decide when to invoke operators
- Scheduling policy may influence **queue sizes**
- **FIFO**: fully process tuple batches in the order of arrival
- **Greedy**: invoke operator discarding most tuples
- **Mix**: combine operators into chains
  - FIFO scheduling within chain, greedy across chains

Scheduling Example

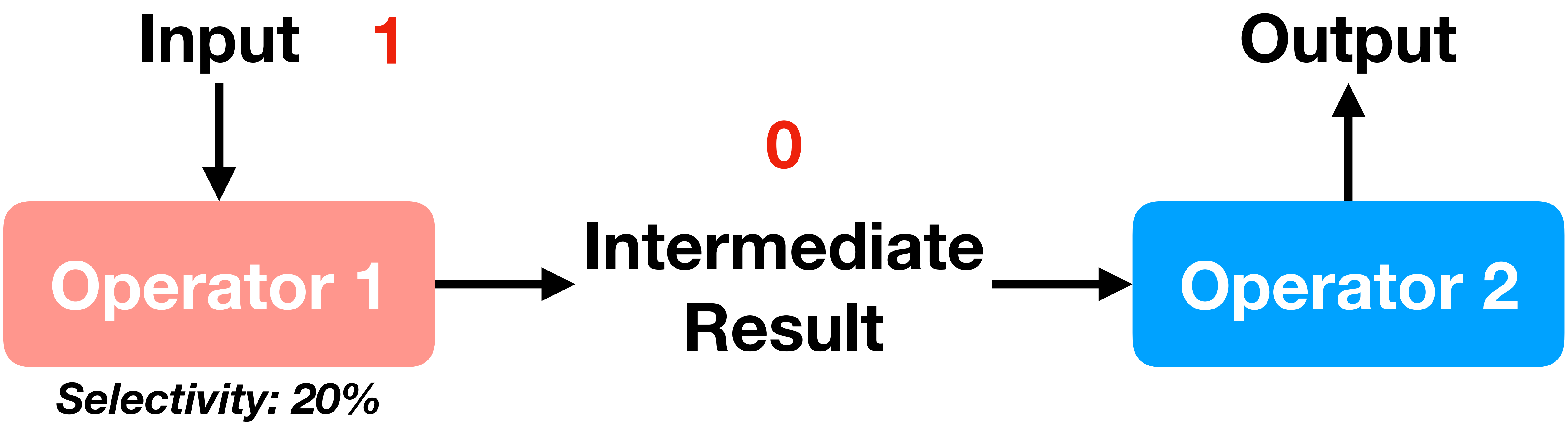
Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO							
Greedy							

Scheduling Example



Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO							
Greedy							

Scheduling Example



Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1						
Greedy							

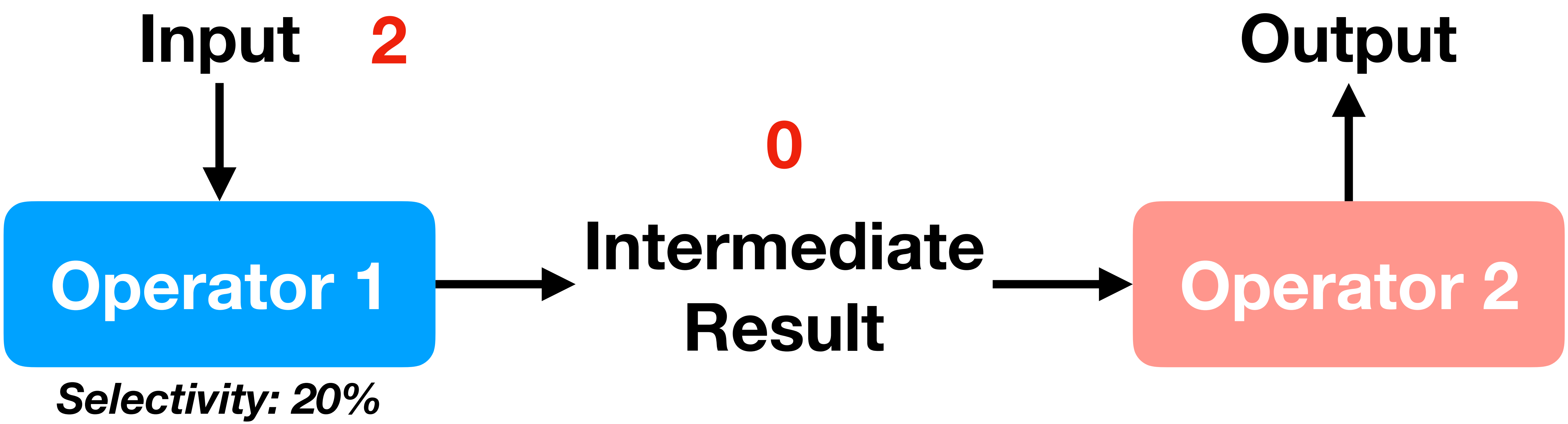
Scheduling Example

Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2					
Greedy							

Scheduling Example

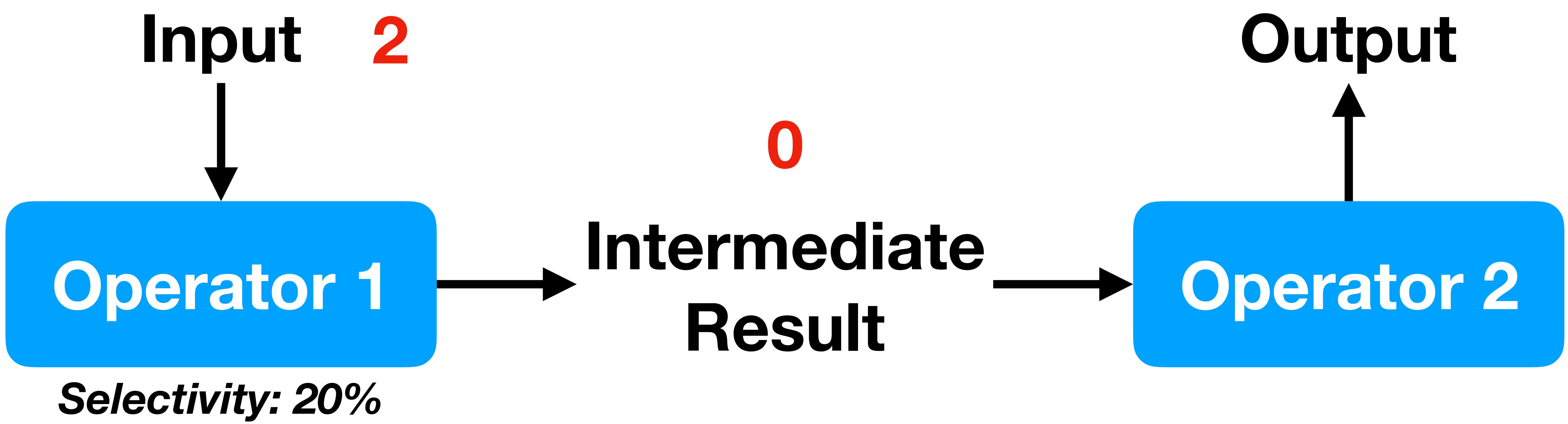
Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2					
Greedy							

Scheduling Example



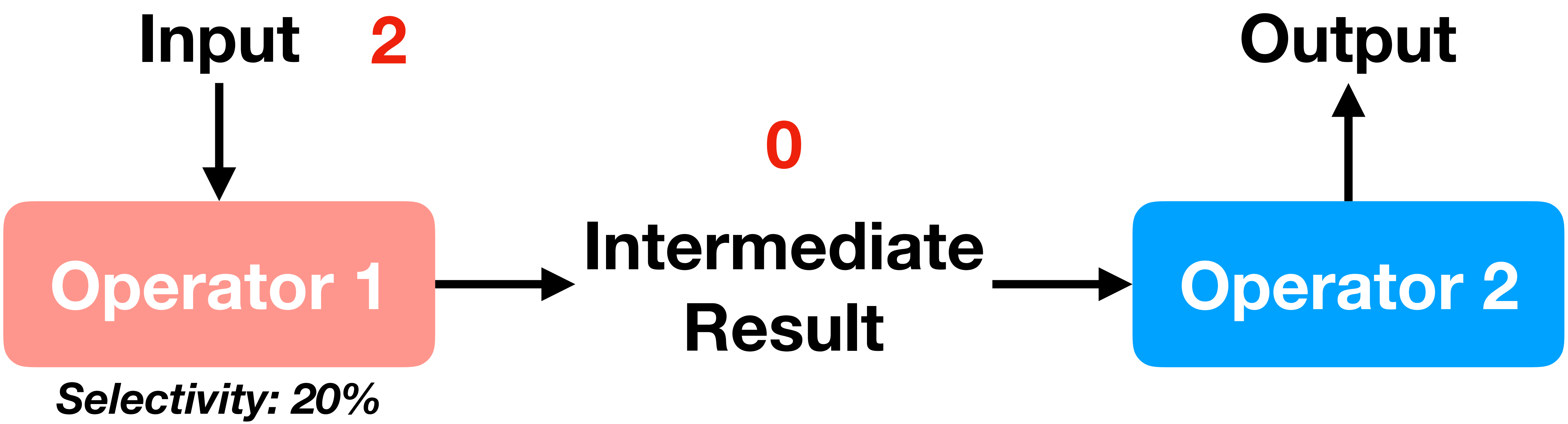
Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2				
Greedy							

Scheduling Example



Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2				
Greedy							

Scheduling Example



Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2				
Greedy							

Scheduling Example

Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2				
Greedy							

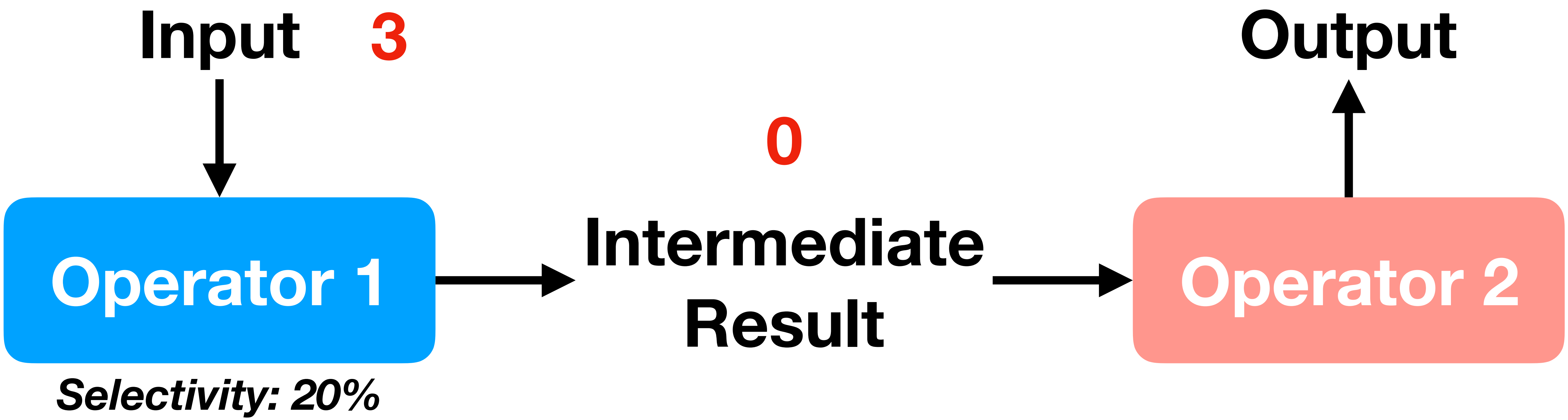
Scheduling Example

Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2			
Greedy							

Scheduling Example

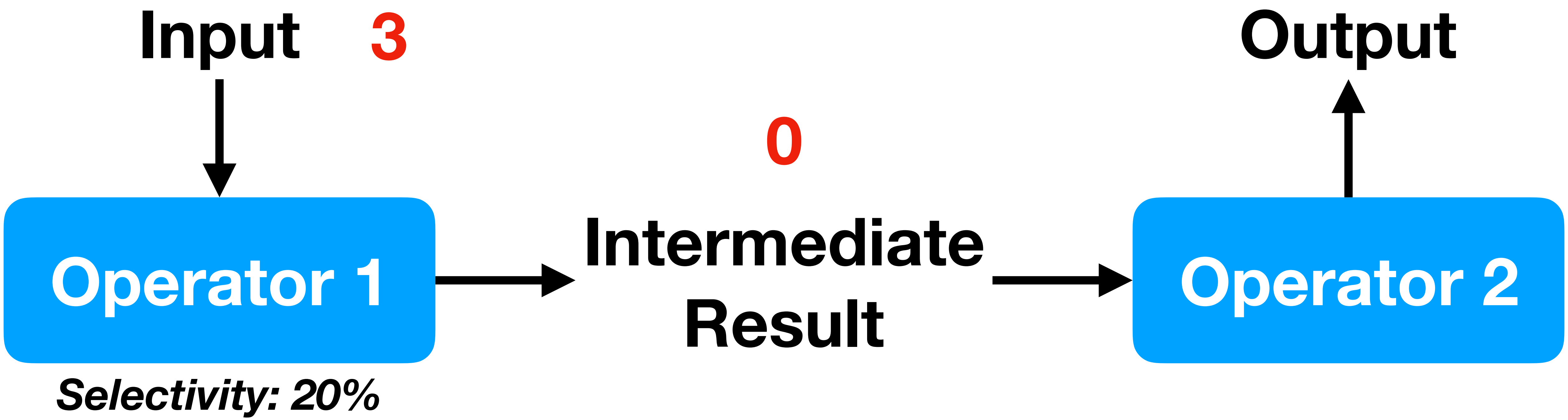
Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2			
Greedy							

Scheduling Example



Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3		
Greedy							

Scheduling Example

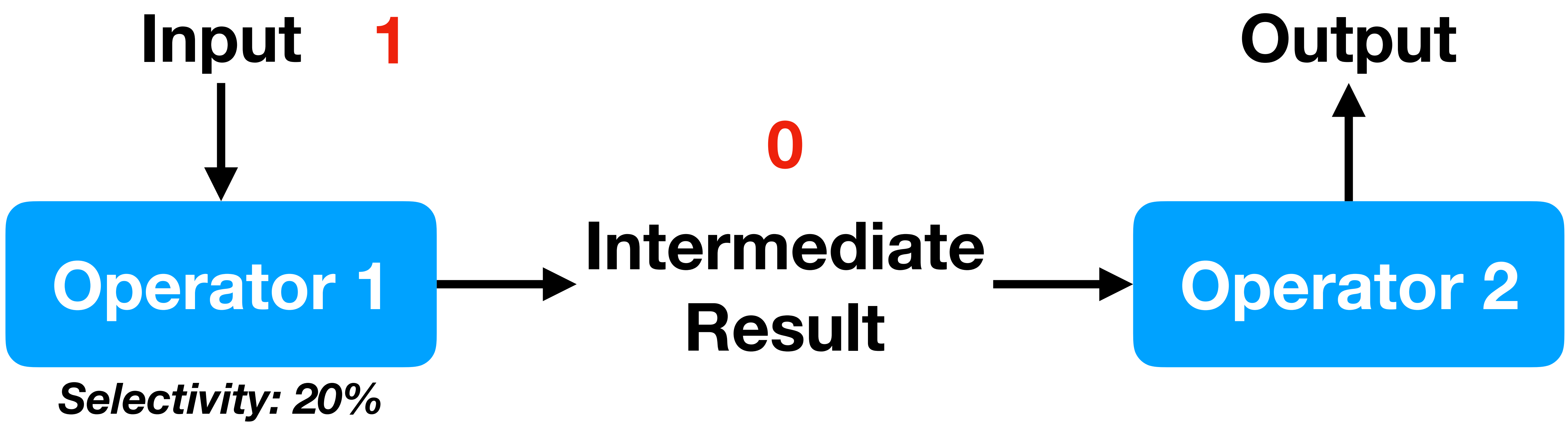


Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3		
Greedy							

Scheduling Example

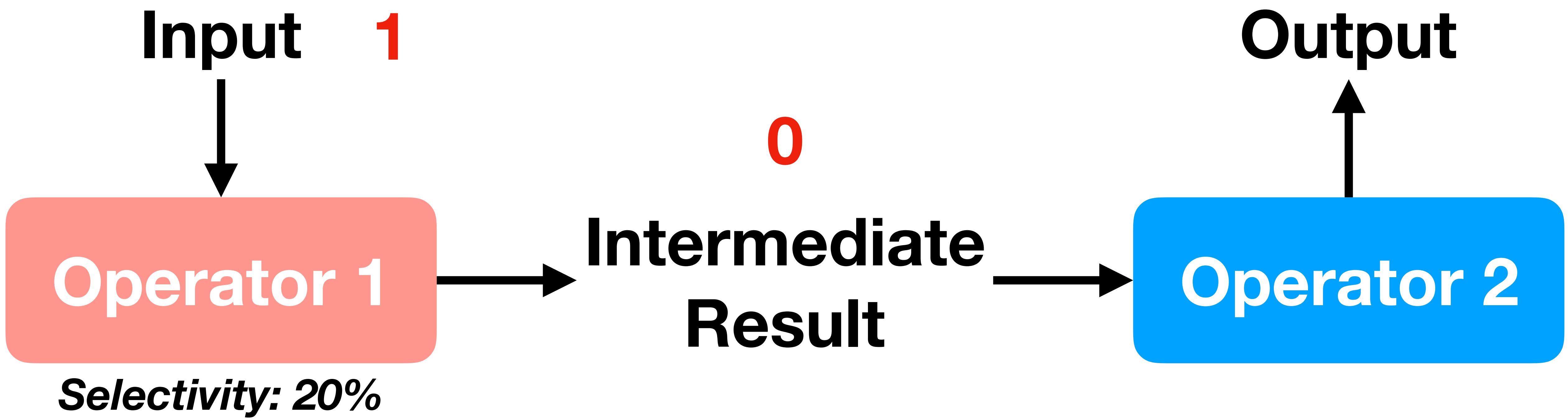
Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3	3.2	4
Greedy							

Scheduling Example



Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3	3.2	4
Greedy	1						

Scheduling Example



Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3	3.2	4
Greedy	1						

Scheduling Example

Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3	3.2	4
Greedy	1	1.2					

Scheduling Example

Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3	3.2	4
Greedy	1	1.2					

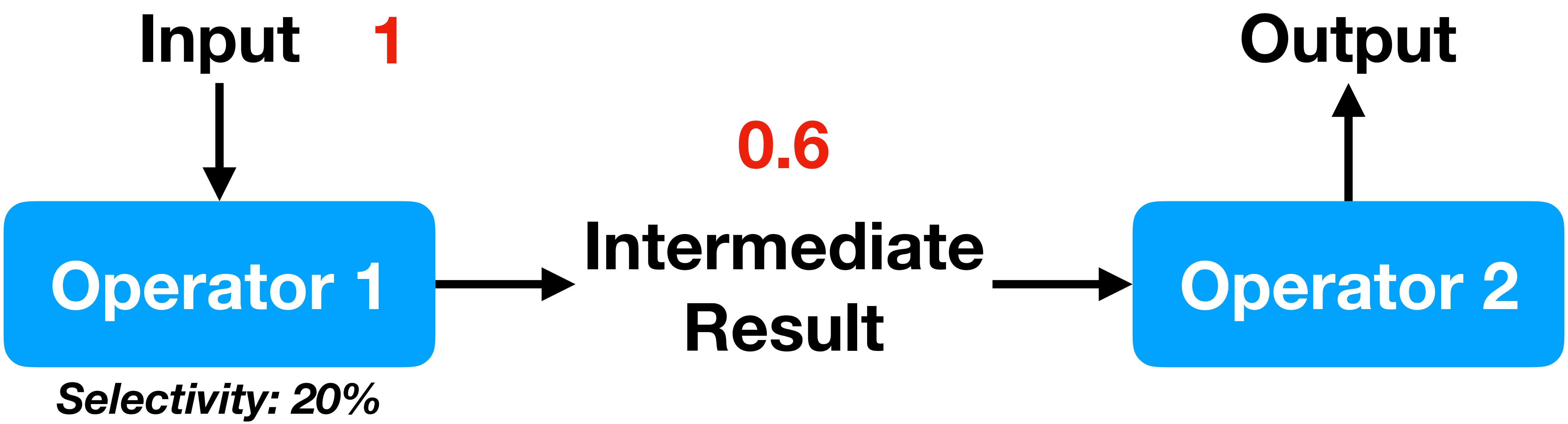
Scheduling Example

Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3	3.2	4
Greedy	1	1.2	1.4				

Scheduling Example

Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3	3.2	4
Greedy	1	1.2	1.4				

Scheduling Example



Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3	3.2	4
Greedy	1	1.2	1.4	1.6			

Scheduling Example

Policy	T=0	T=1	T=2	T=3	T=4	T=5	T=6
FIFO	1	1.2	2	2.2	3	3.2	4
Greedy	1	1.2	1.4	1.6	1.8	2	2.2

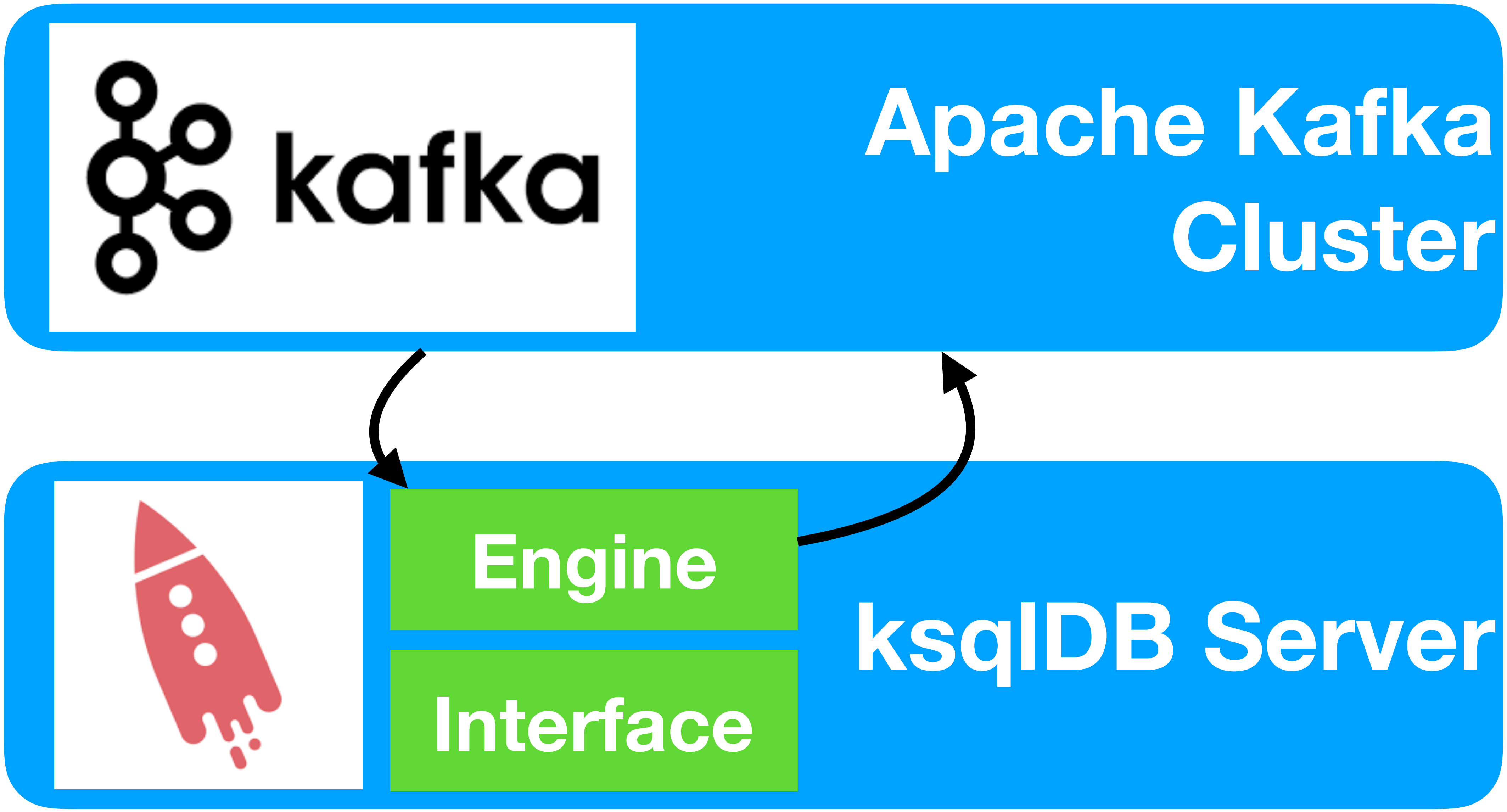
Approximation

- **Load shedding**: drop tuples to save overheads
  - Can approximate aggregates based on samples
  - Try to balance impact over all aggregates
- **Reducing synopses** sizes: save memory
  - Often reduces output size of following operators
    - ***Are there any exceptions … ?***

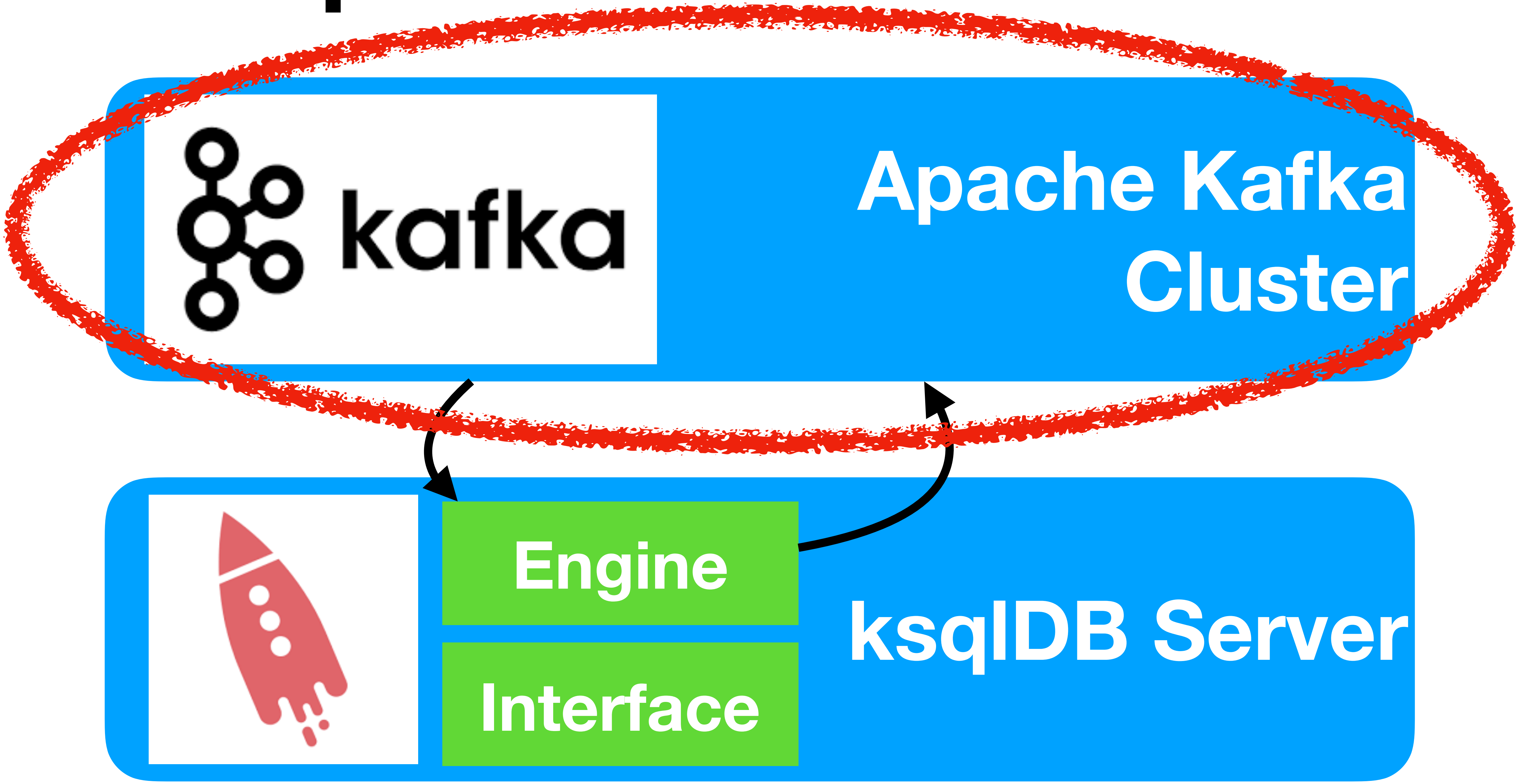
Data Stream Topics

- STREAM System (~2003)
  - First "Stream Data Management System"
- **ksqlDB (~2020)**
  - Recent system for distributed stream processing

ksqlDB Architecture



ksqlDB Architecture



Apache Kafka Overview

- A Java-based, distributed **stream processing** engine
- **Producers** can add records to different topics
- **Consumers** can subscribe to specific topics
- Kafka Streams API offers filter/grouping/... **operators**
- E.g., used by **Uber** for passenger-driver matching

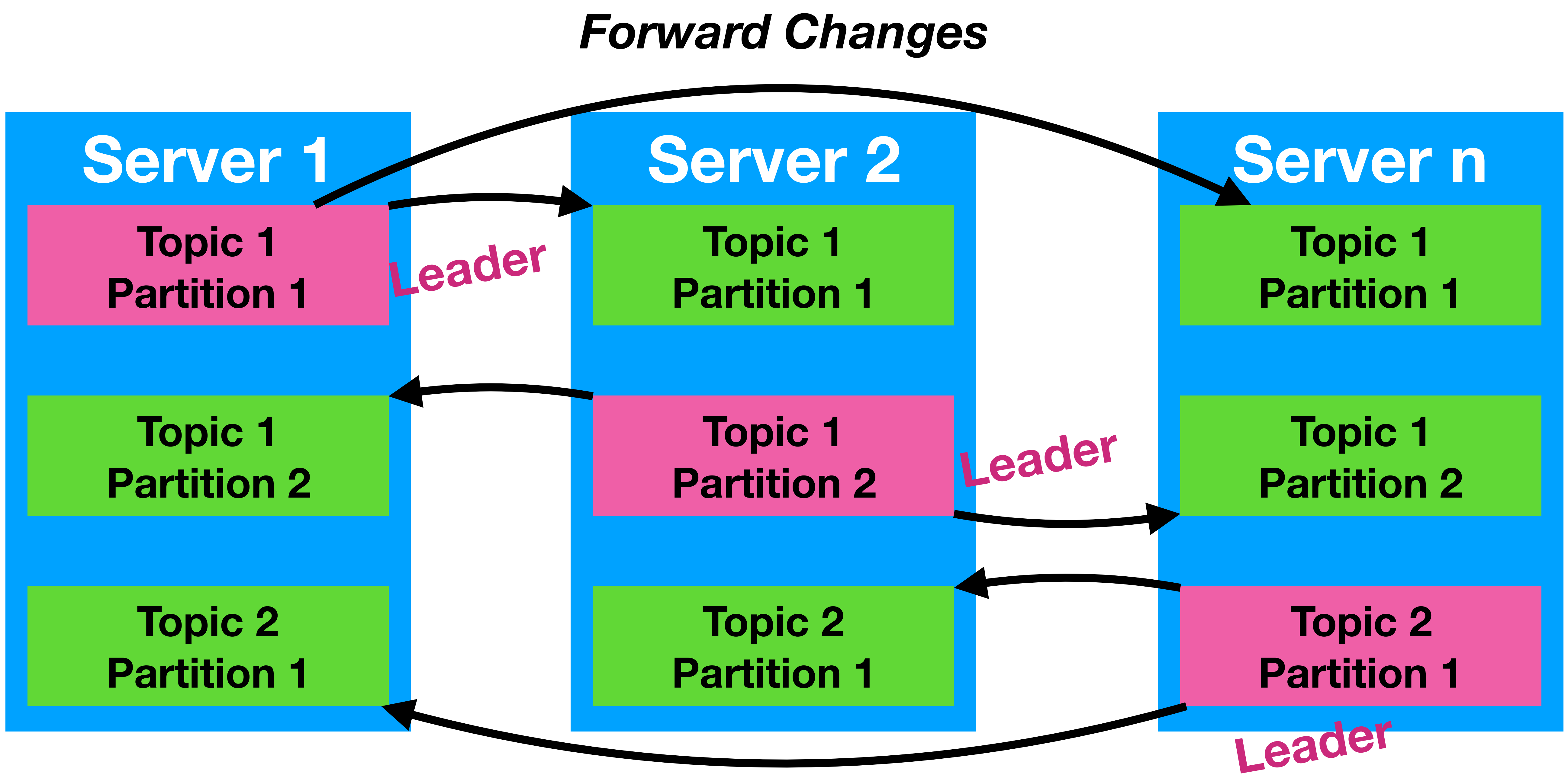
Kafka Topics

- Each topic corresponds to a log of **ordered records**
    - Each record is a **key-value** pair
- **Producers** append to this log - no updates/deletes!
- **Consumers** receive updates for topics they subscribed to
- **Regular** topic: delete tuples by space/time constraint
- **Compacted** topic: new tuples override old keys

Distributed Processing

- Each topic is divided into **partitions**
- Partitions are **replicated** across servers
  - **Fault tolerance** by redundancy
  - Allows to **scale** to more consumers
- Each partition has one dedicated **leader**
  - Leader accepts topics **updates**
  - **Synchronizes** with other replicas

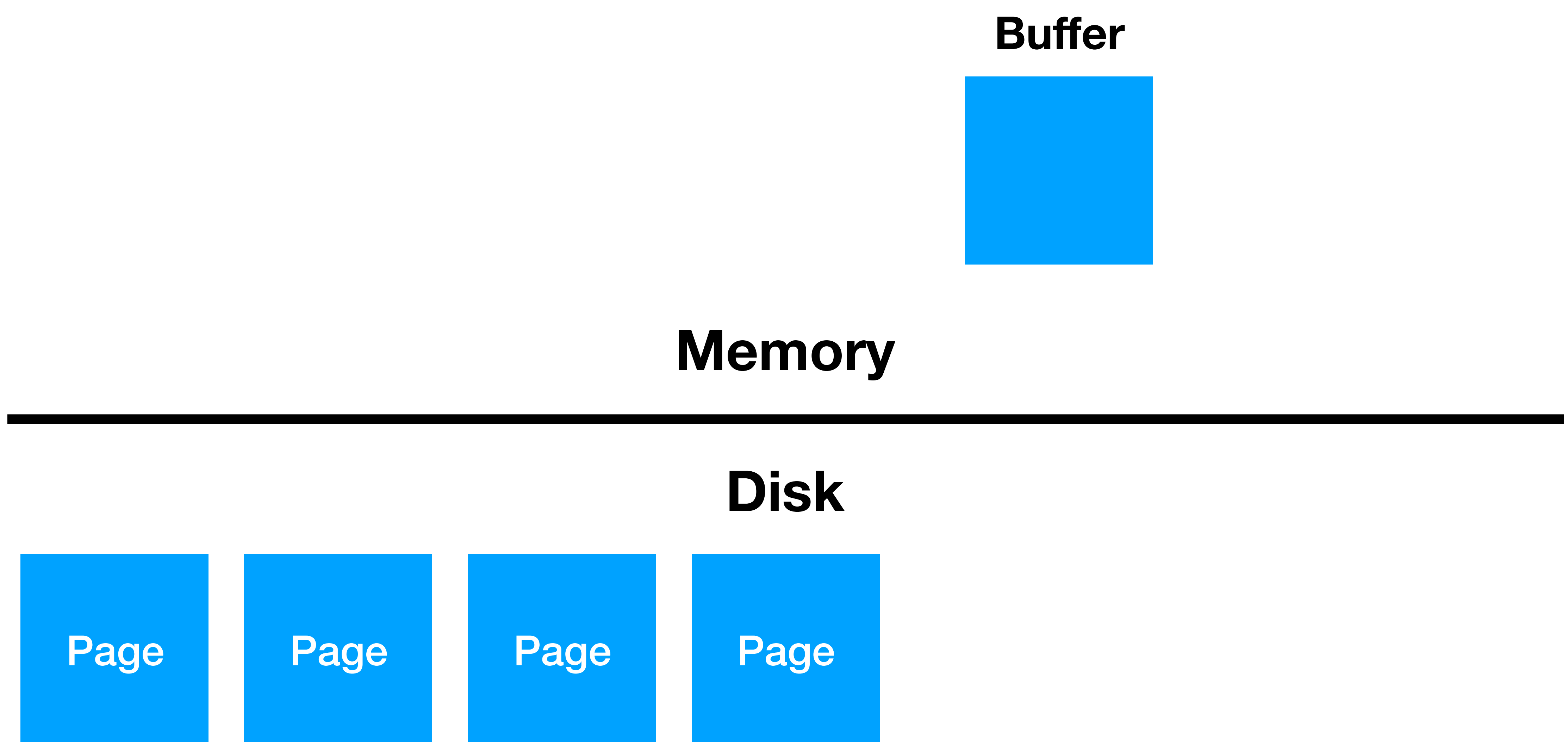
Distributed Processing



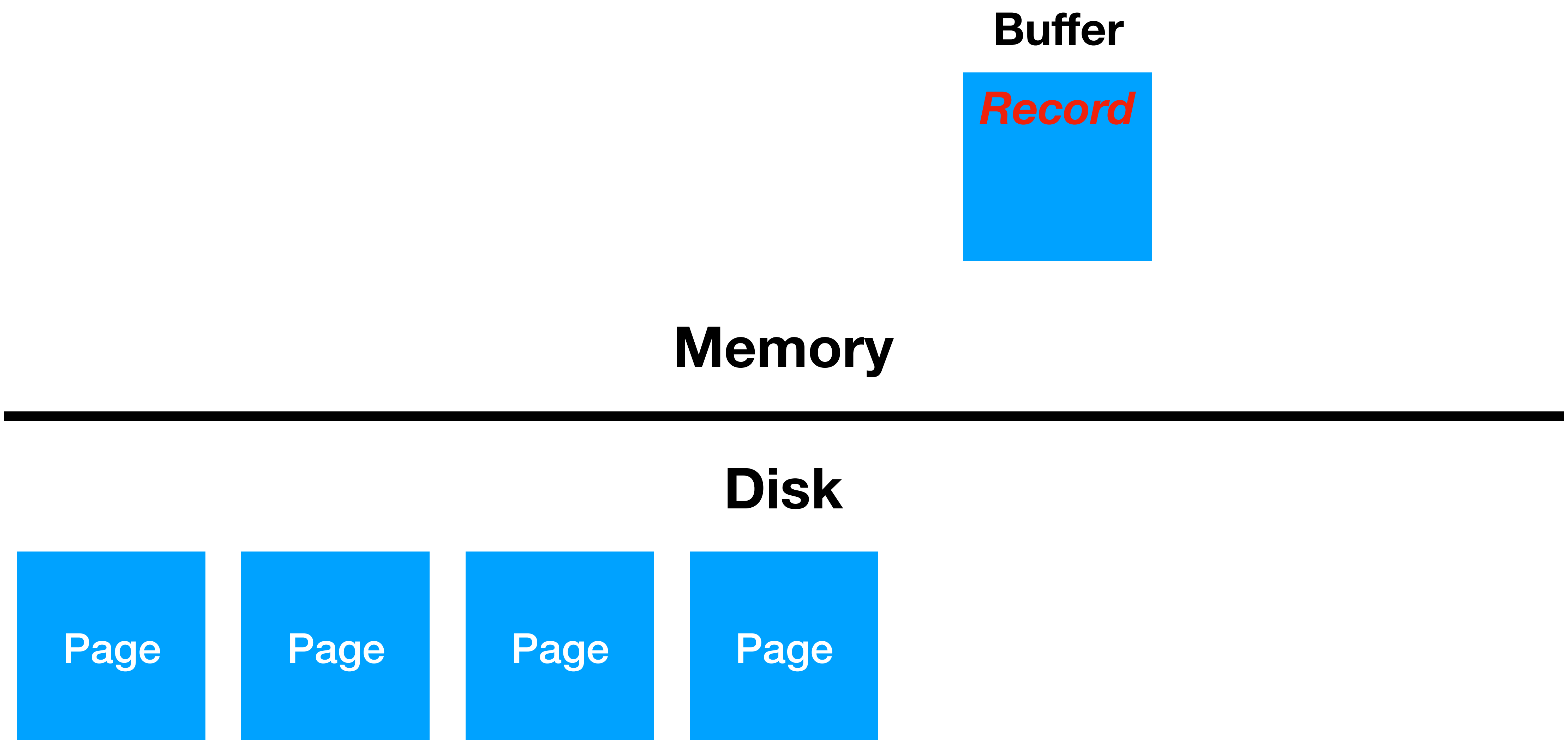
Coping with Insertions

- Need to handle insertions with a **very high** frequency
- Kafka Streams uses **RocksDB** as underlying engine
- Highly **optimized for writes**, good read performance
    - Key idea: **sequential** (instead of random) access

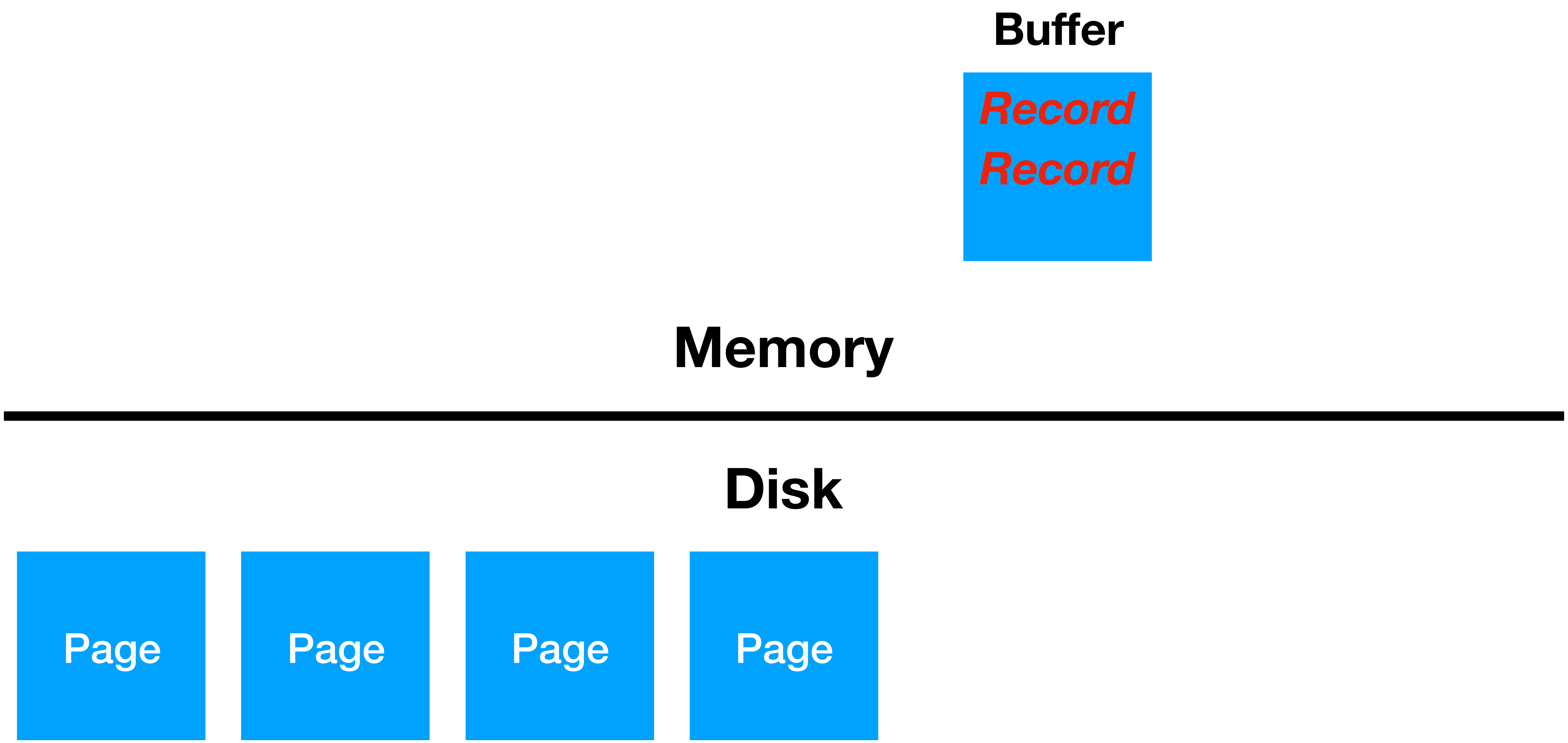
Optimize for Insertions



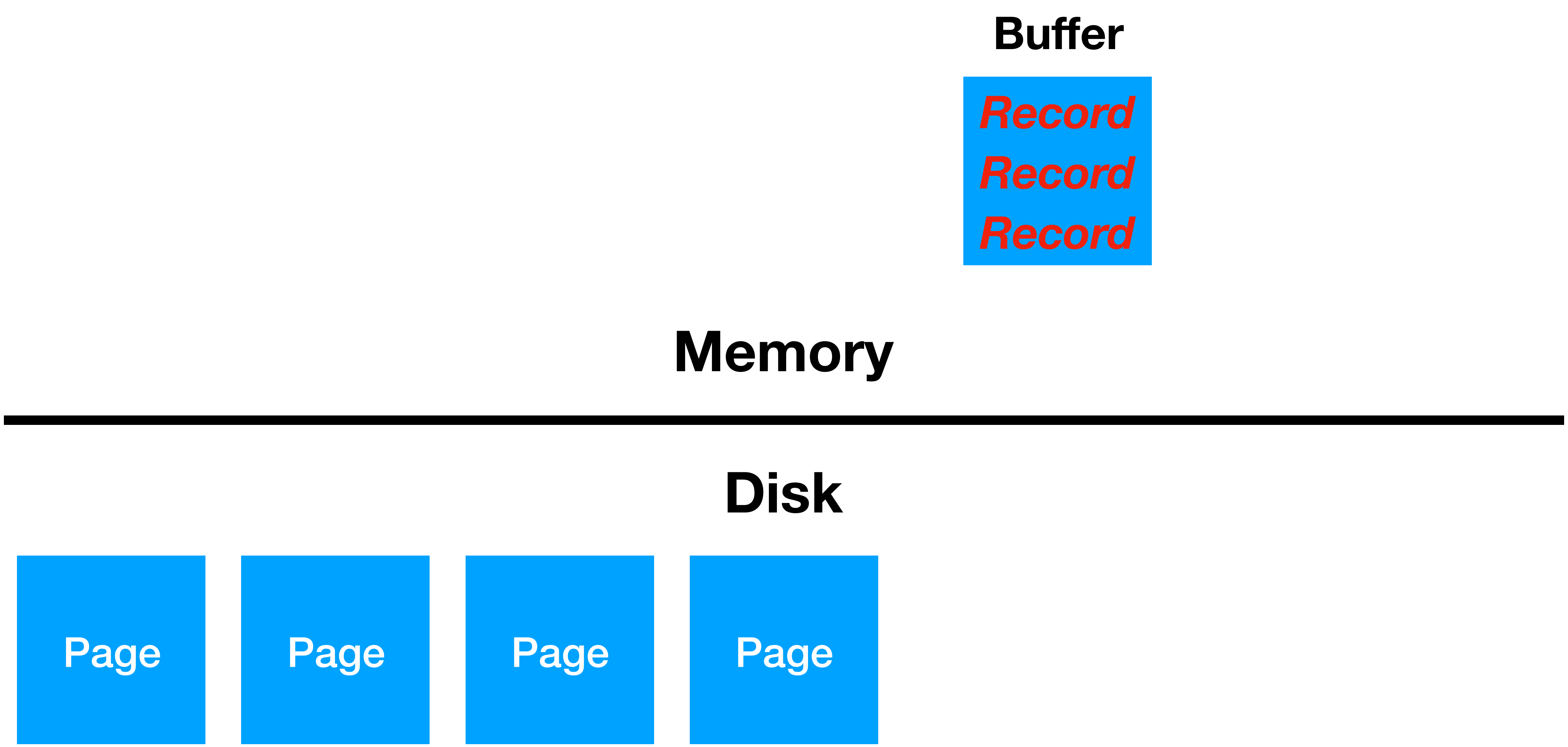
Optimize for Insertions



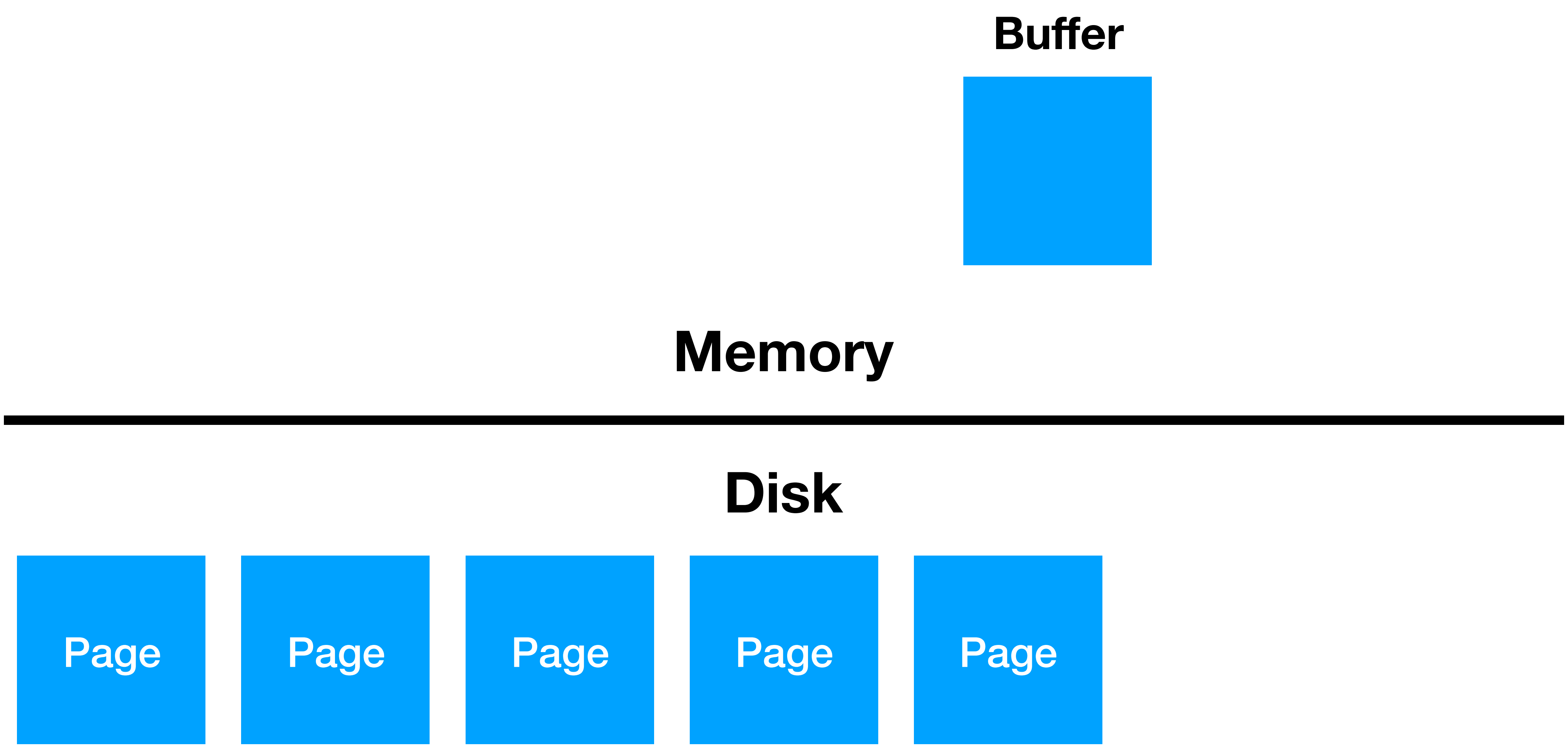
Optimize for Insertions



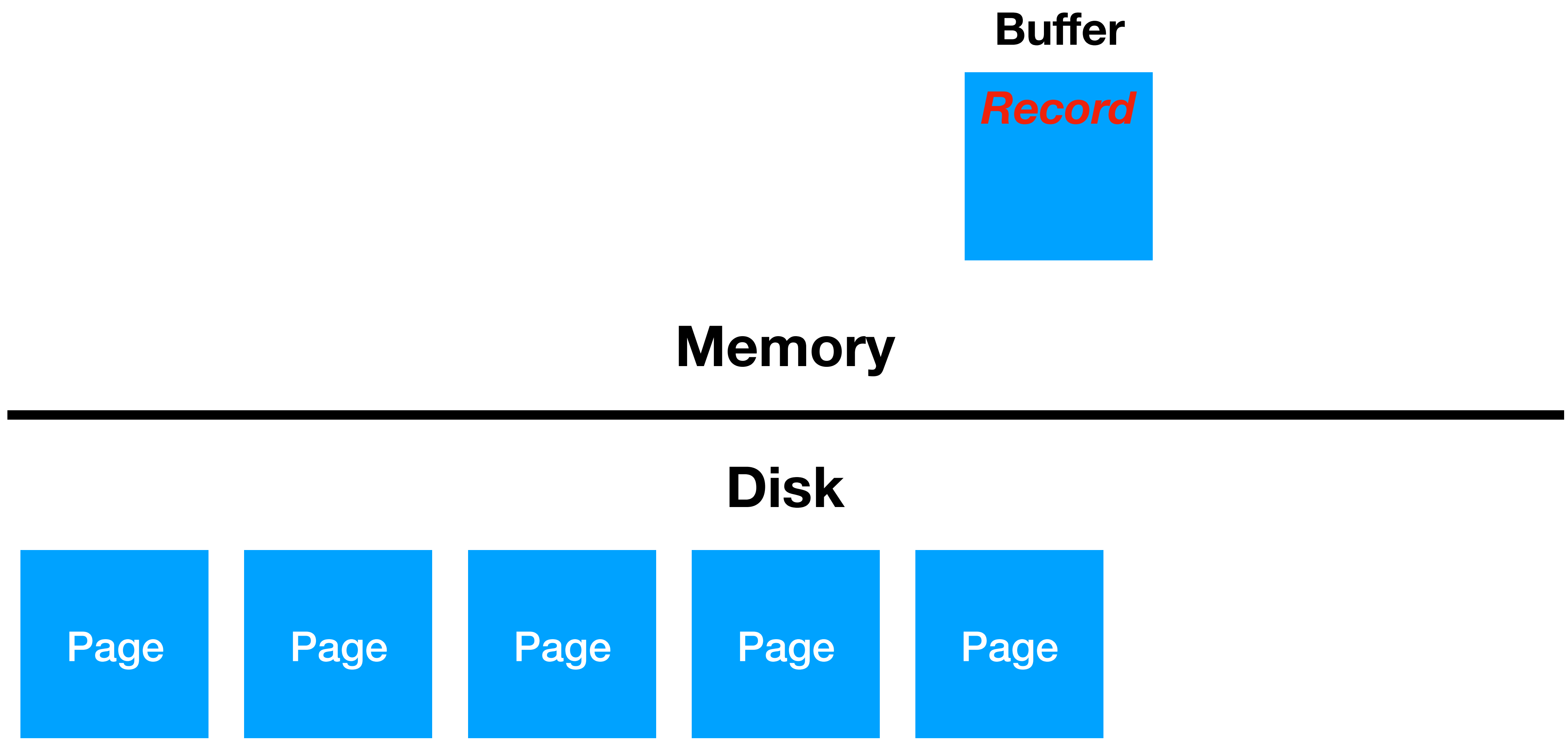
Optimize for Insertions



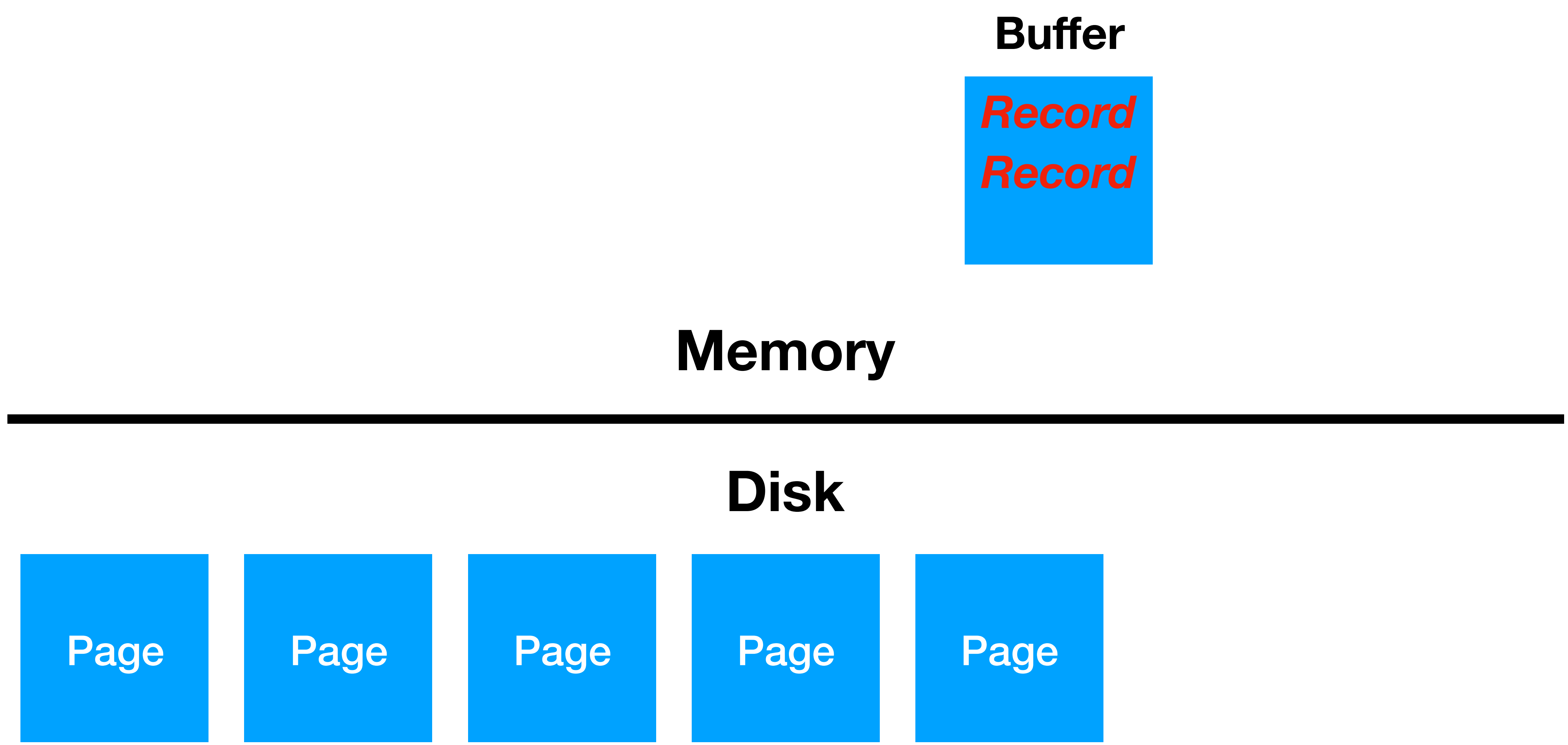
Optimize for Insertions



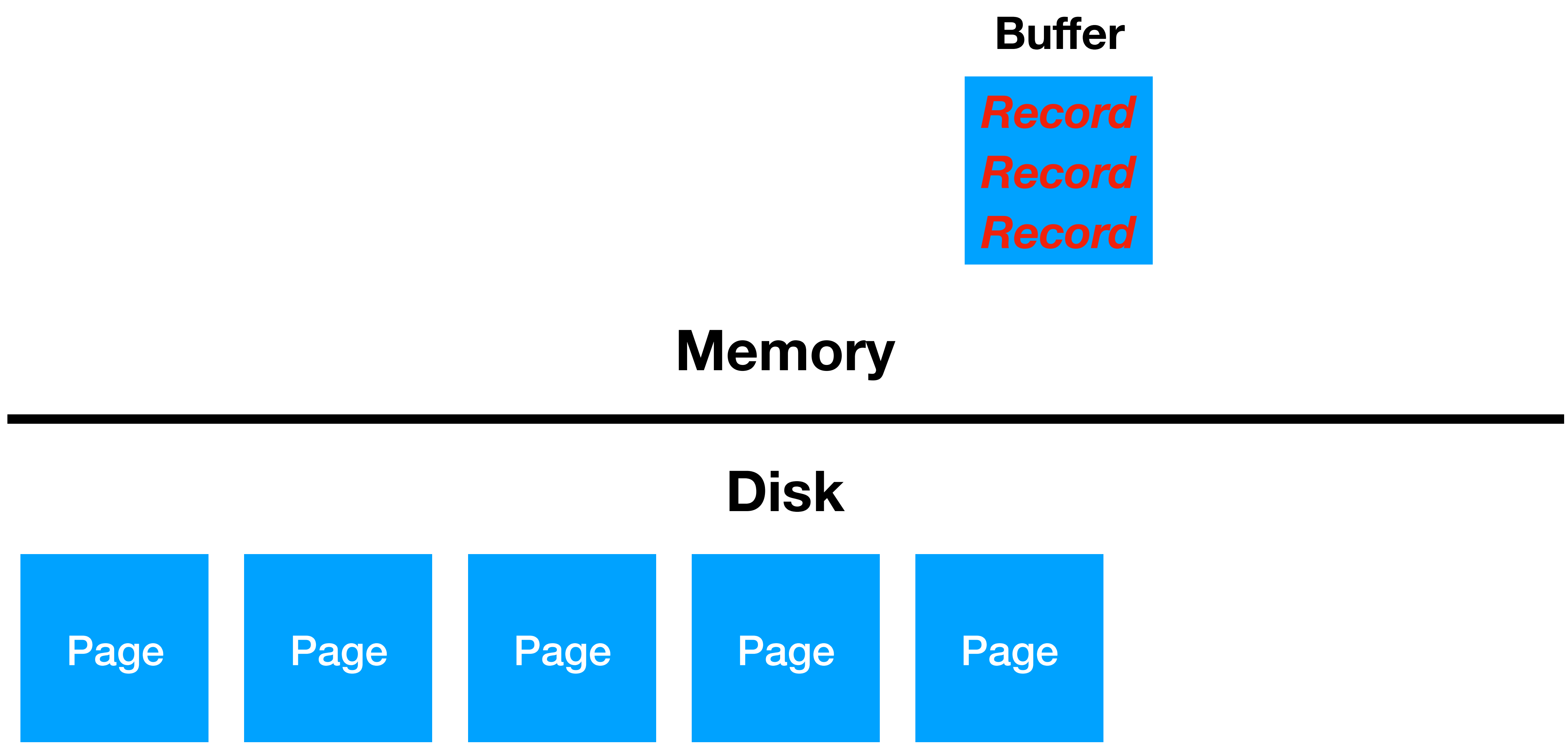
Optimize for Insertions



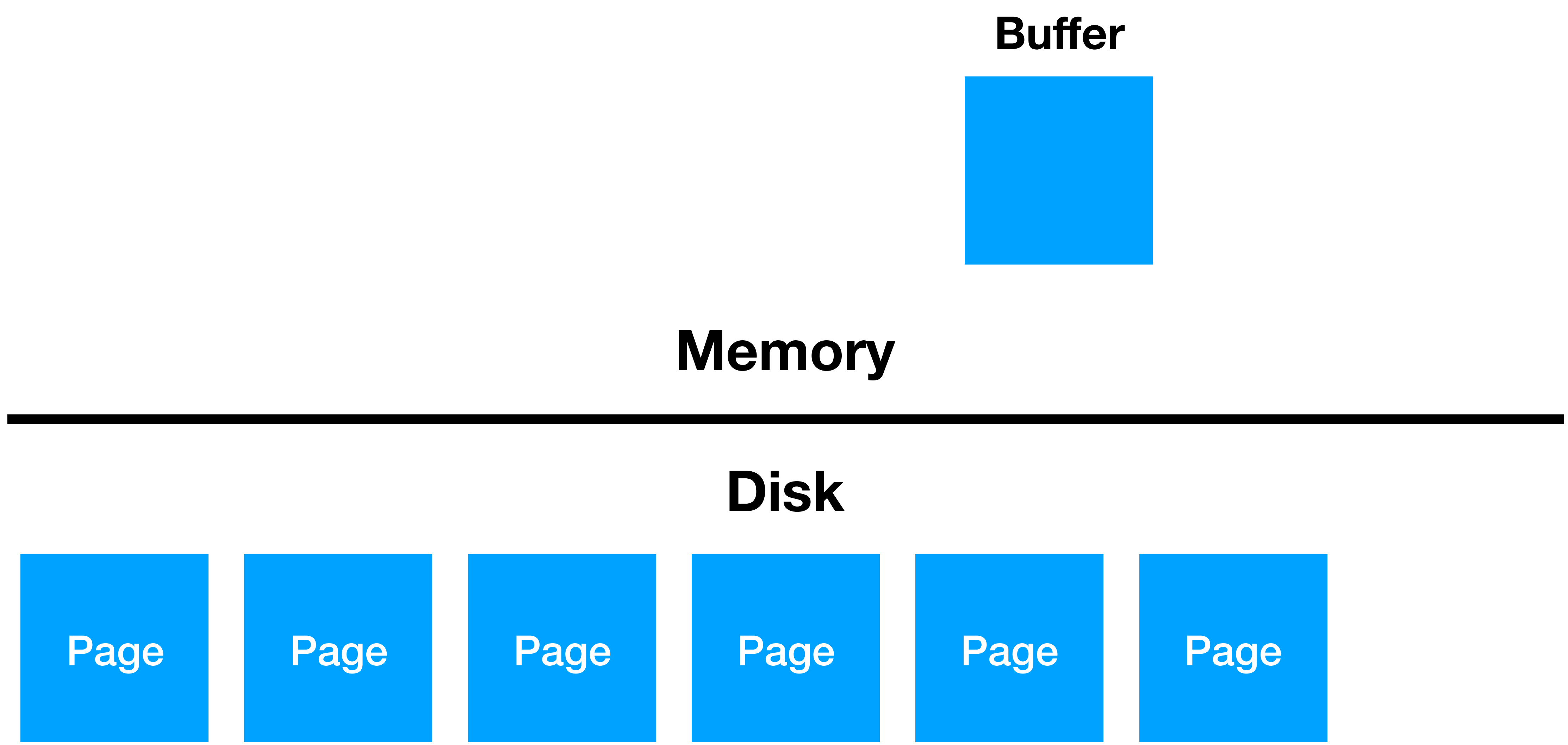
Optimize for Insertions



Optimize for Insertions

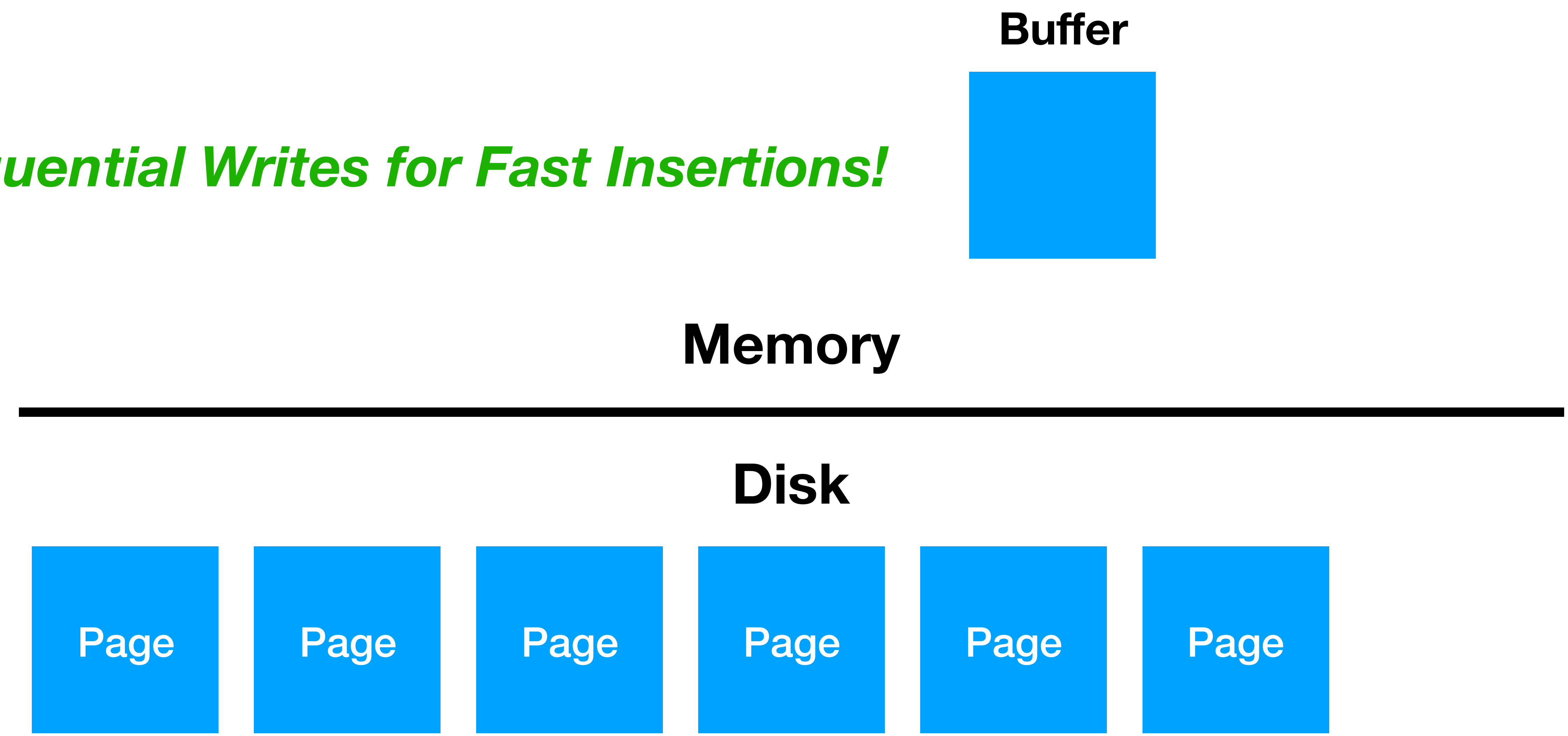


Optimize for Insertions



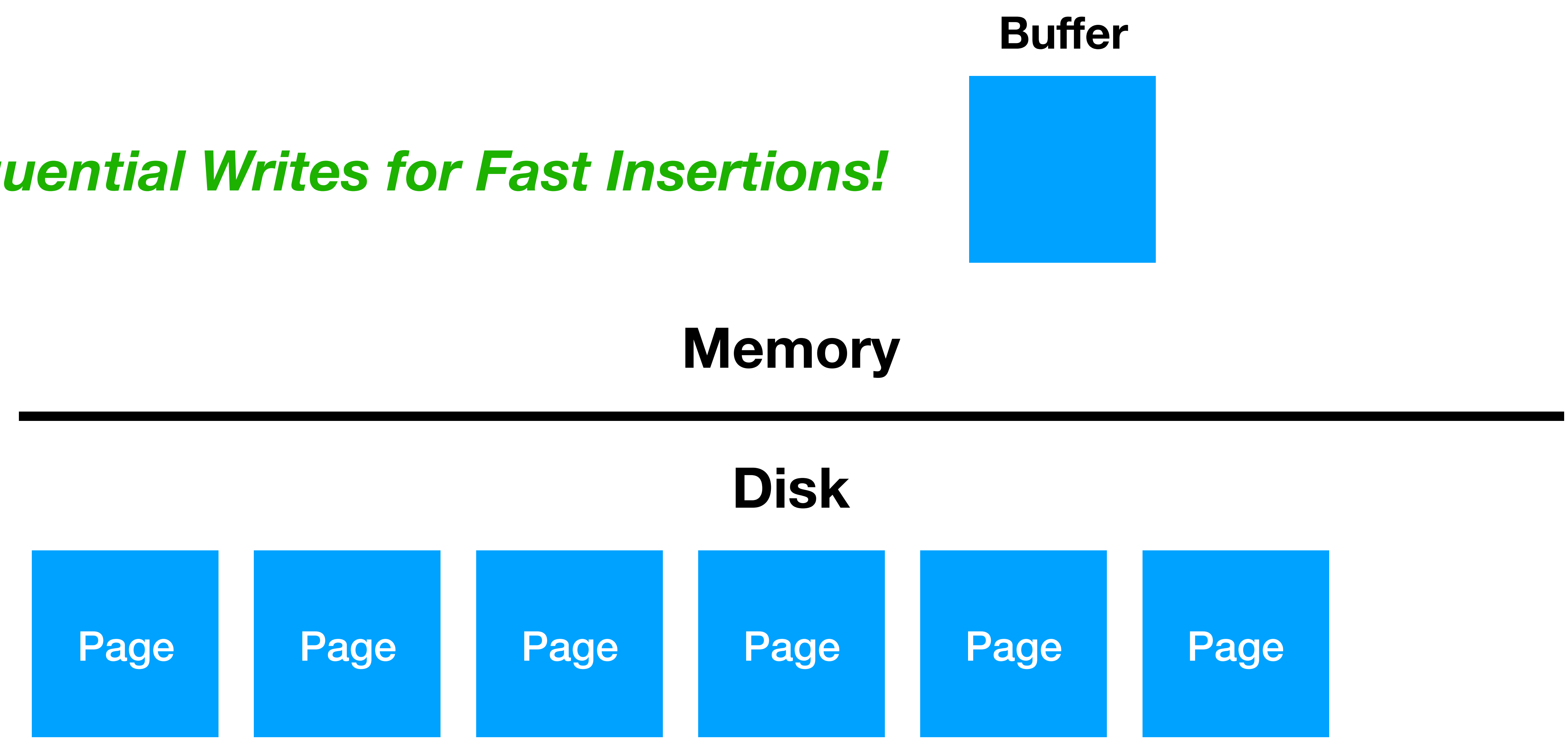
Optimize for Insertions

Sequential Writes for Fast Insertions!



Optimize for Insertions

Sequential Writes for Fast Insertions!

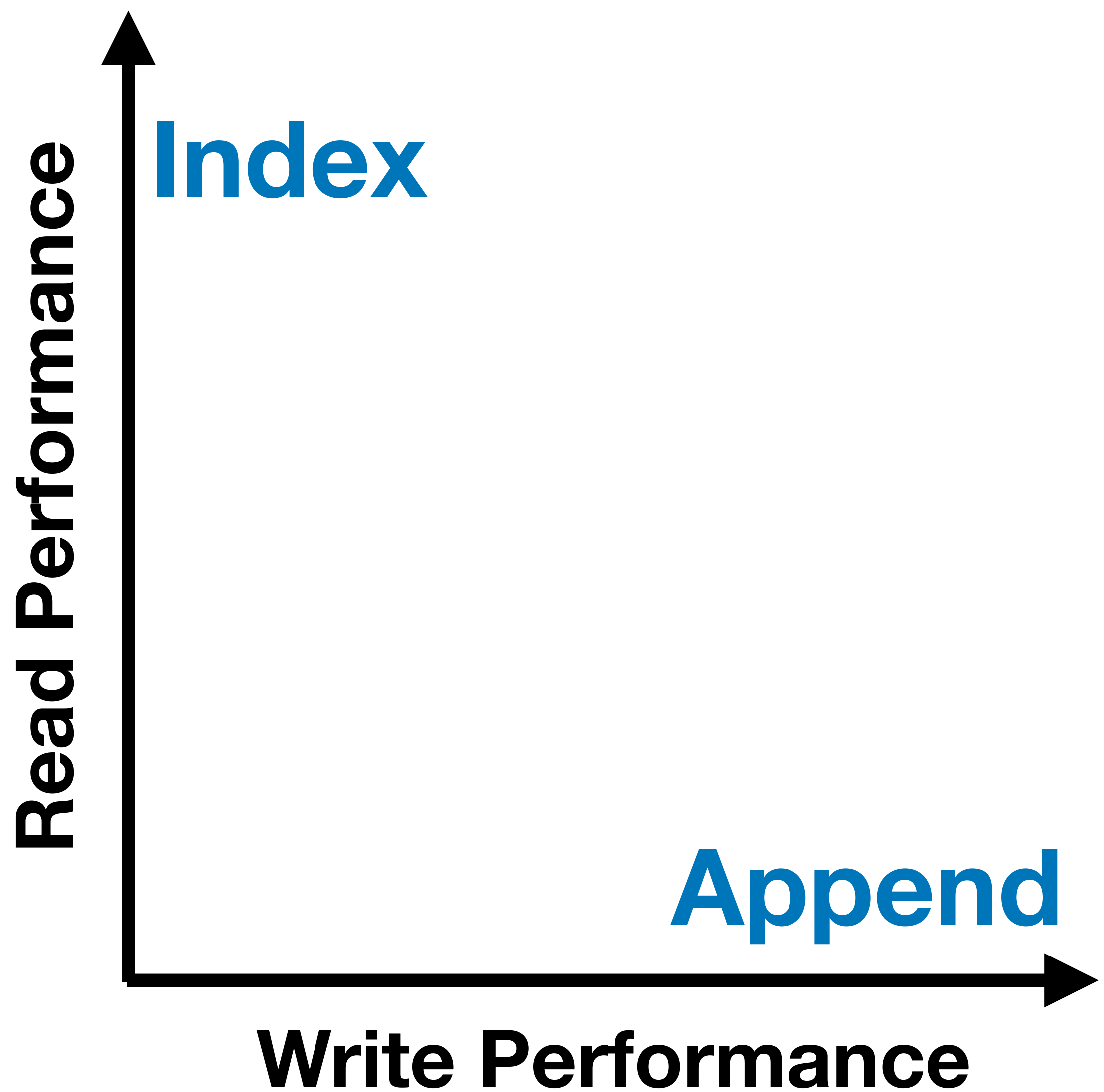


Need to Read Everything to Find Specific Key!

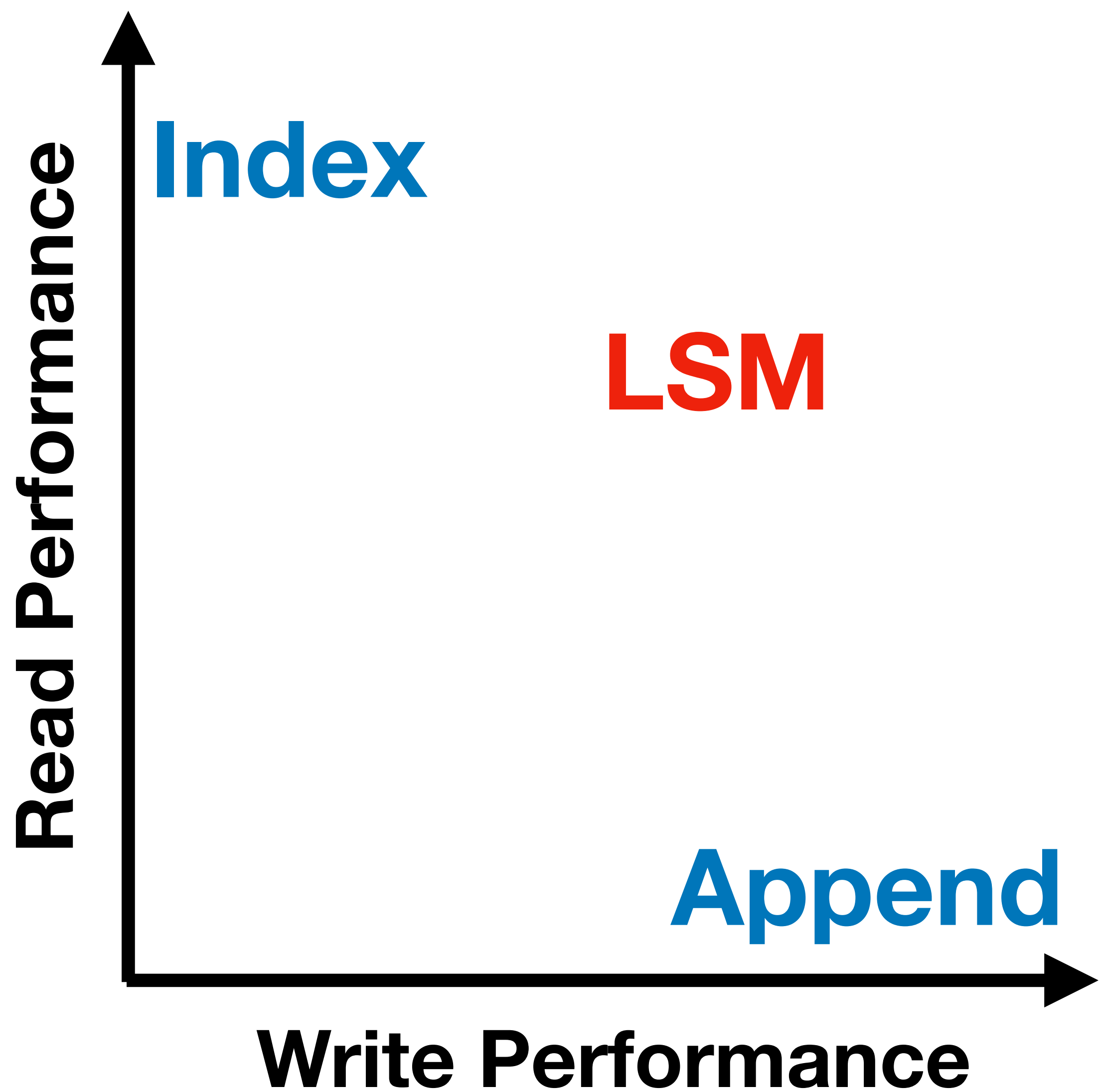
Optimize for Reads

- Typically use **index structure** to speed up reads
  - E.g., **B+ tree** seen previously in class
- But then insertions require **random** data access
- Leads to **slow insertions** - not acceptable for streams!

Read vs. Write Performance



Read vs. Write Performance



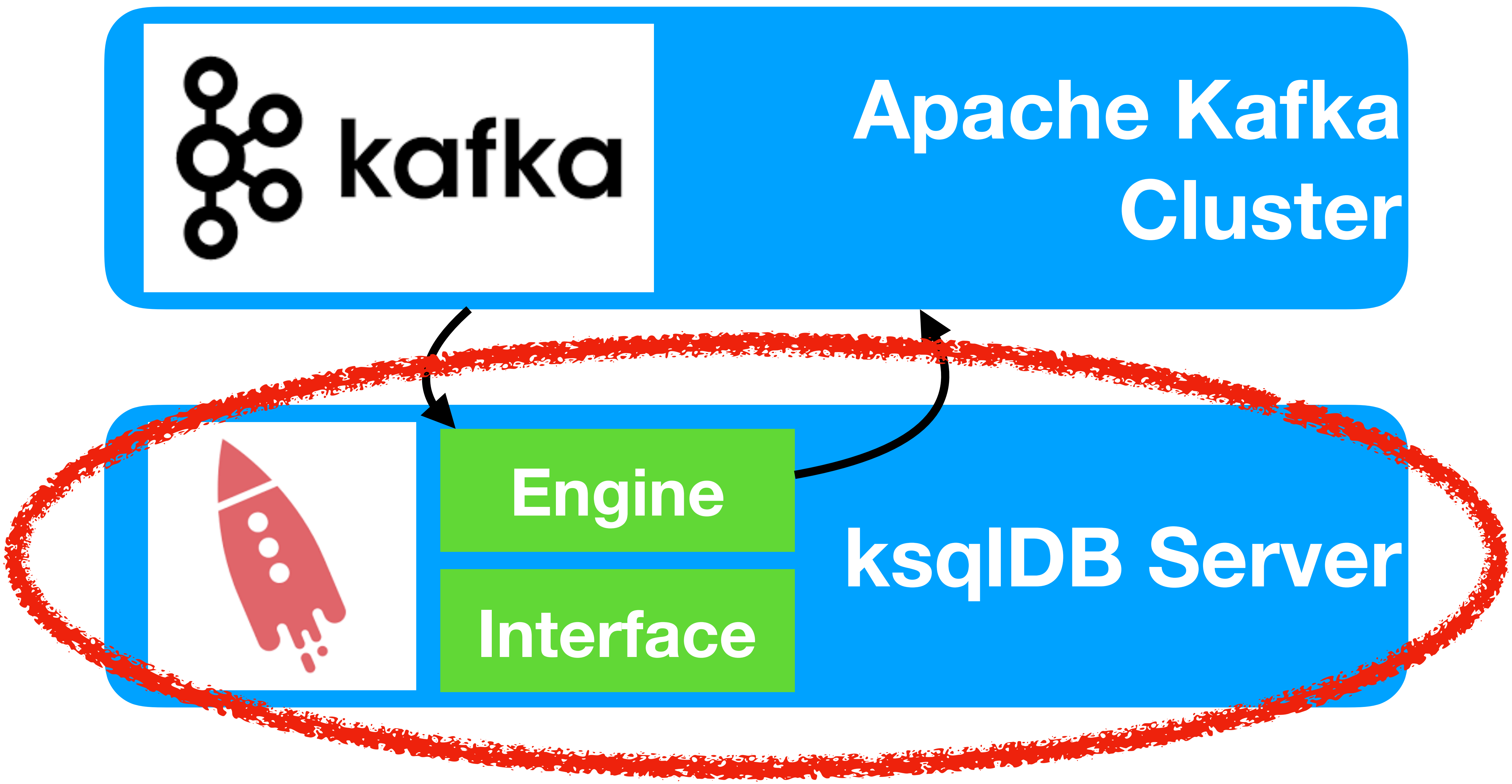
Log Structured Merge Tree (with Leveling Merge Policy)

- Maintains **multiple levels** containing sorted/indexed data
  - Upper level(s) are stored in **main memory**
  - Lower levels are stored on **hard disk**
  - Constant **size ratio** between consecutive levels
- Data from one level is **merged** into next at overflow
  - Merge operations need only **sequential** writes

Reading LSM Trees

- May have to check **every level** to find data
- Checking each level is **fast** as data is sorted/indexed
- **Bloom filters** reduce the number of levels to consider
  - (We have seen Bloom filters for distributed joins!)
  - Bloom filter captures non-empty hash buckets
  - Used to summarize keys present at each level

ksqlDB Architecture



ksqlDB

- **High-level API** on top of Kafka Streams
- Translates **SQL-like queries** to Kafka operators
  - Some similarities to STREAM query language
- Processes **collections** of events: streams and tables
- **Pull queries** execute once on current state
- **Push query** results get continuously updated

ksqlDB Collection Types

	Stream	**Table**
Insertion semantics	New entries are appended	New entries override prior entries with same key
Purpose	Represent historical information	Represent the current state

Creating Collections

- **CREATE STREAM** priceHistory(symbol varchar, price int)
  WITH (kafka_topic = 'tickerTopic', value_format = 'JSON')

- **CREATE TABLE** curStockPrice(
    symbol varchar **PRIMARY KEY**, price int)
  WITH (kafka_topic = 'tickerTopic', value_format = 'JSON')

Creating Collections

- **CREATE STREAM** priceHistory(symbol varchar, price int)
  WITH (kafka_topic = 'tickerTopic', value_format = 'JSON')

- **CREATE TABLE** curStockPrice(
    symbol varchar **PRIMARY KEY**, price int)
  WITH (kafka_topic = 'tickerTopic', value_format = 'JSON')

Need to associated with Kafka topic!

Deriving Collections

- **CREATE STREAM** appleTicker **AS**
  SELECT * FROM priceHistory WHERE symbol = 'AAPL'

- **CREATE STREAM** advertisementStream **AS**
  SELECT * FROM clickStream C JOIN advertiserTable A
  ON C.advertiserID = A.advertiserID

Inserting Data

- **INSERT**
  INTO temperatureStream (Location, temperature)
  VALUES ('Ithaca', 32)

Query Types

	Push Query	Pull Query
Data Sources	Table, Stream	Table
Specific Restrictions	-	Non-windowed aggregation: lookup by key
Life Time	Keeps returning updates	Returns one result

Query Examples

- **Pull Query:**
  SELECT * FROM pageviewsByRegionTable
    WHERE region = 'Ithaca'

- **Push Query:**
  SELECT * FROM clickEventStream
    WHERE region = 'Ithaca'
    EMIT Changes

(Demo)

Streams Summary

- Systems that analyze **data streams** in real time
- Motivates extensions to the **SQL** query language
- Need to keep **memory consumption** low
- May use specialized **data structures** for fast inserts
- **Distributed** stream processing required to scale