ETRI

Electronics and Telecommunications
Research Institute

Parallel Data Processing

Parallel Query Execution: Volcano and Morsel-Driven Parallelism

Superstar Database #13

15 April 2016

Sung-Soo Kim

sungsoo@etri.re.kr

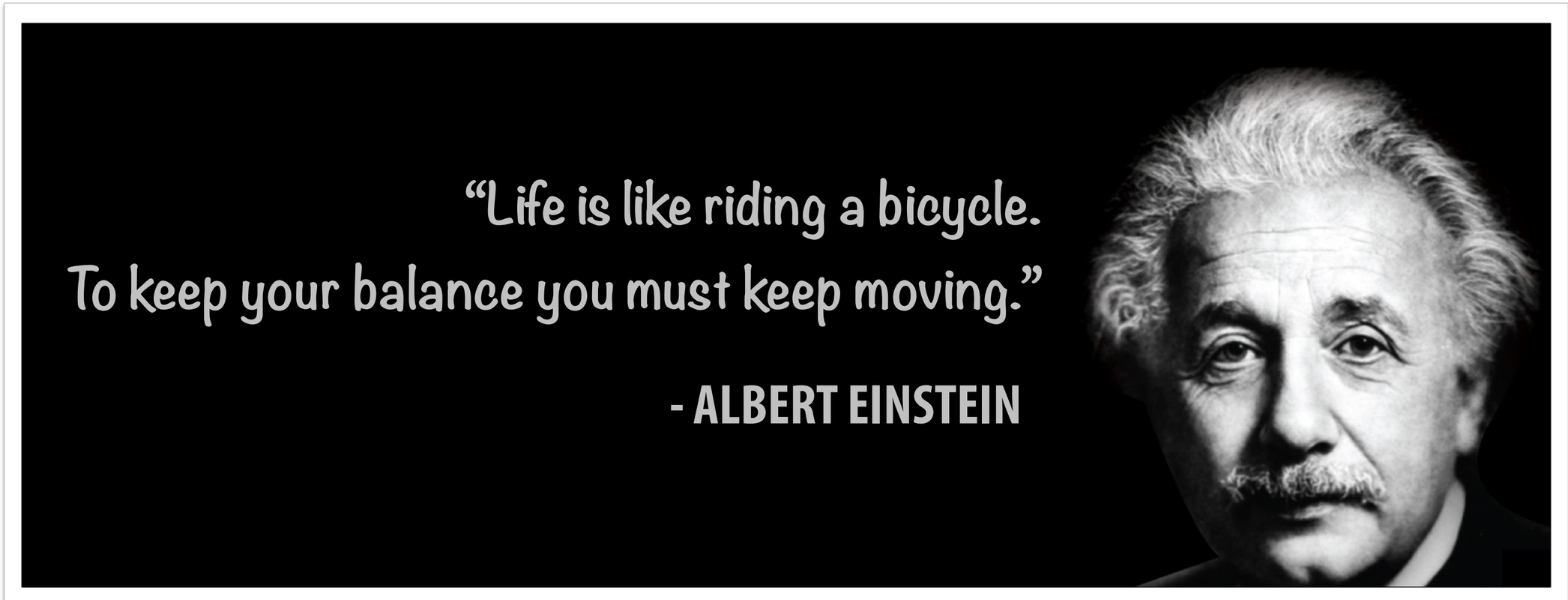
Data Platform Research Section

ETRI


QUESTIONS?
COMMENTS?
CONCERNS?

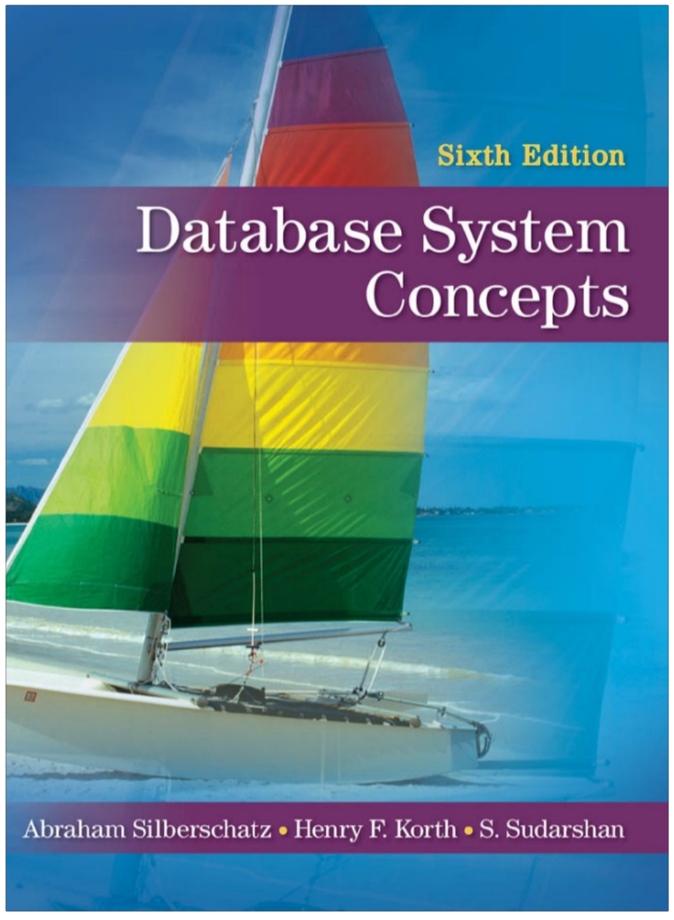
Outline

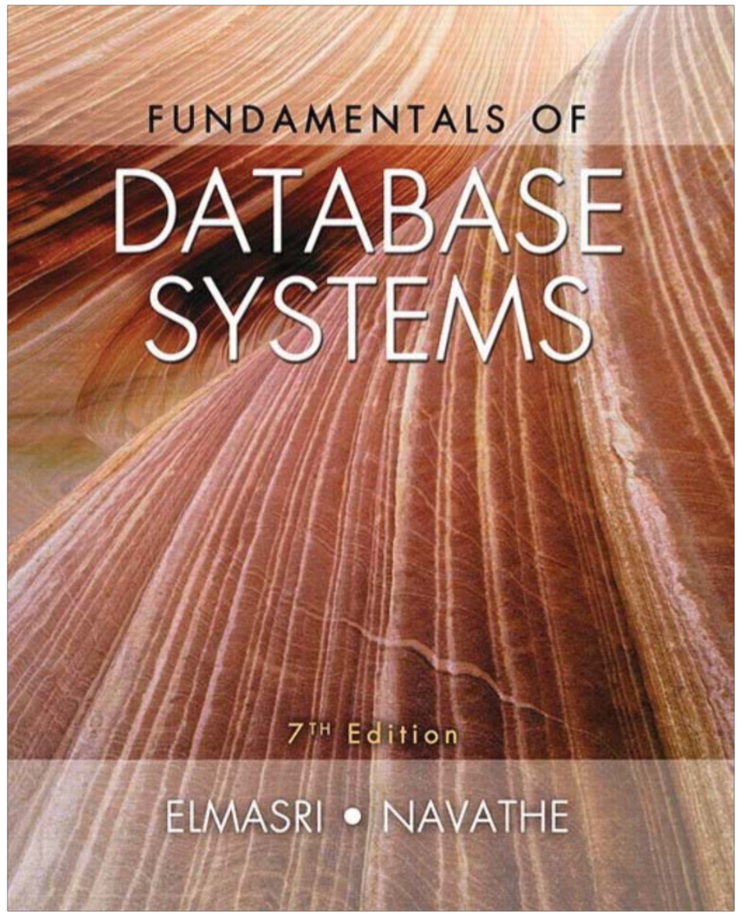
- Motivation: Start with "Why Parallelism?"
- Amdahl's Law and Gusfafson's Law
- Parallel Programming Models [~~MapReduce~~]
- Parallelism in In-Memory Databases
- Parallel Execution Models: Volcano and Morsel-Driven Parallelism
- Summary



References & Slide Credits

- *Database System Concepts, 6th Edition*
- *Fundamentals of Database Systems, 7th Edition*
- Volcano-An Extensible And Parallel Query Evaluation System, *IEEE TKDE 1994*
- Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age, *SIGMOD 2014*





Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Viktor Leis* Peter Boncz† Alfons Kemper* Thomas Neumann*

* Technische Universität München † CWI

* {leis,kemper,neumann}@in.tum.de † p.boncz@cwi.nl

ABSTRACT

With modern computer architecture evolving, two problems conspire against the state-of-the-art approaches in parallel query execution: (i) to take advantage of many-cores, all query work must be distributed evenly among (soon) hundreds of threads in order to achieve good speedup, yet (ii) dividing the work evenly is difficult even with accurate data statistics due to the complexity of modern out-of-order cores. As a result, the existing approaches for "plan-driven" parallelism run into load balancing and context-switching bottlenecks, and therefore no longer scale. A third problem faced by many-core architectures is the decentralization of memory controllers, which leads to Non-Uniform Memory Access (NUMA).

In response, we present the "morsel-driven" query execution framework, where scheduling becomes a fine-grained run-time task that is NUMA-aware. Morsel-driven query processing takes small fragments of input data ("morsels") and schedules these to worker threads that run entire operator pipelines until the next pipeline breaker. The degree of parallelism is not baked into the plan but can elastically change during query execution, so the dispatcher can react to execution speed of different morsels but also adjust resources dynamically in response to newly arriving queries in the workload. Further, the dispatcher is aware of data locality of the NUMA-local morsels and operator state, such that the great majority of executions takes place on NUMA-local memory. Our evaluation on the TPC-H and SSB benchmarks shows extremely high absolute performance and an average speedup of over 30 with 32 cores.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing

Keywords

Morsel-driven parallelism; NUMA-awareness

1. INTRODUCTION

The main impetus of hardware performance improvement nowadays comes from increasing multi-core parallelism rather than from speeding up single-threaded performance [2]. By SIGMOD 2014

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
ACM 978-1-4503-2376-5/14/06.
http://dx.doi.org/10.1145/2588555.2610507 .

Figure 1: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$

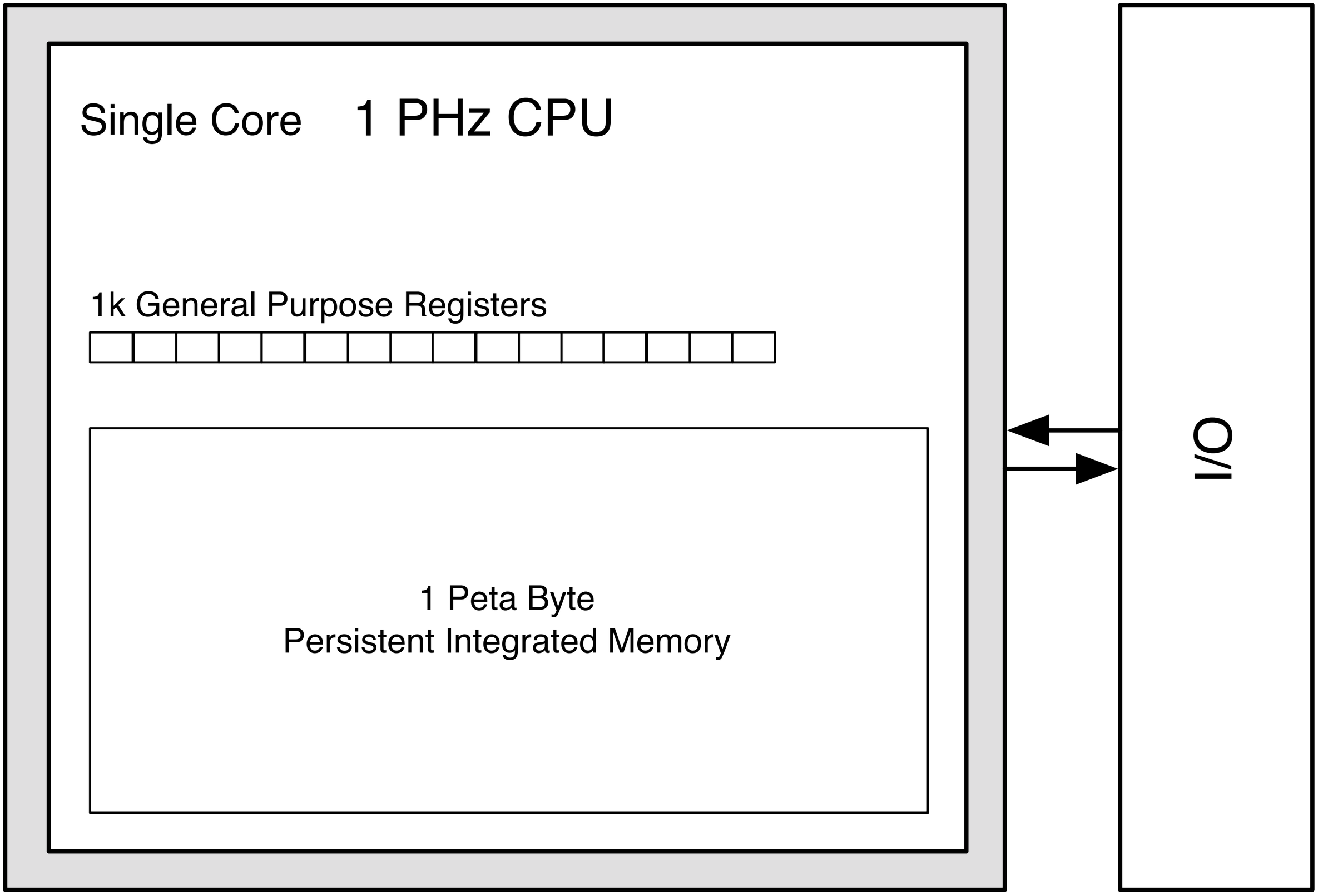
Intel's forthcoming mainstream server Ivy Bridge EX, which can run 120 concurrent threads, will be available. We use the term *many-core* for such architectures with tens or hundreds of cores.

At the same time, increasing main memory capacities of up to several TB per server have led to the development of main-memory database systems. In these systems query processing is no longer I/O bound, and the huge parallel compute resources of many-cores can be truly exploited. Unfortunately, the trend to move memory controllers into the chip and hence the decentralization of memory access, which was needed to scale throughput to huge memories, leads to non-uniform memory access (NUMA). In essence, the computer has become a network in itself as the access costs of data items varies depending on which chip the data and the accessing thread are located. Therefore, many-core parallelization needs to take RAM and cache hierarchies into account. In particular, the NUMA division of the RAM has to be considered carefully to ensure that threads work (mostly) on NUMA-local data.

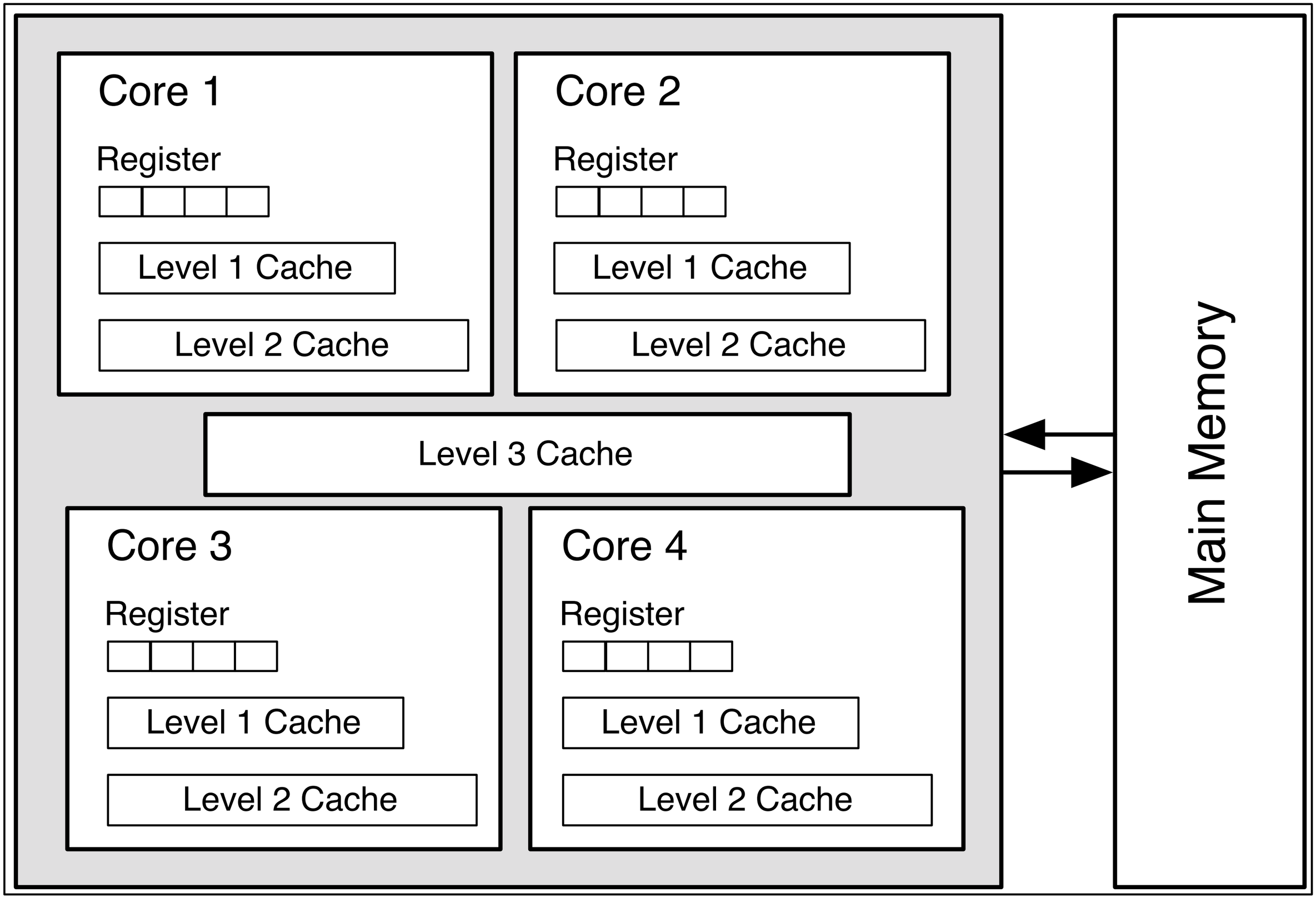
Abundant research in the 1990s into parallel processing led the majority of database systems to adopt a form of parallelism inspired by the Volcano [12] model, where operators are kept largely unaware of parallelism. Parallelism is encapsulated by so-called "exchange" operators that route tuple streams between multiple threads each executing identical pipelined segments of the query plan. Such implementations of the Volcano model can be called *plan-driven*: the optimizer statically determines at query compile-time how many threads should run, instantiates one query operator plan for each thread, and connects these with exchange operators.

In this paper we present the adaptive *morsel-driven* query execution framework, which we designed for our main-memory database system HyPer [16]. Our approach is sketched in Figure 1 for the three-way-join query $R \bowtie_A S \bowtie_B T$. Parallelism is achieved

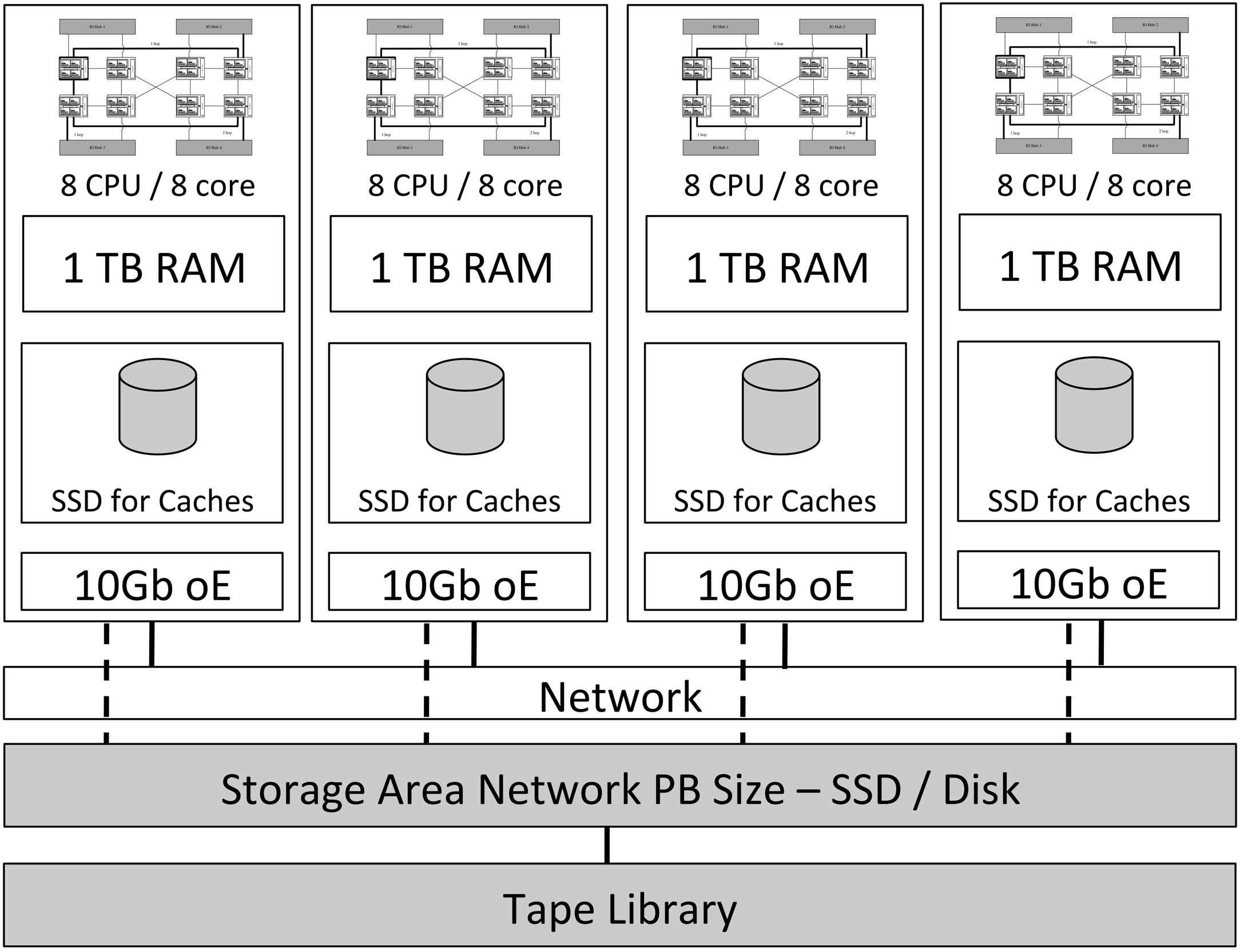
The Ideal Hardware?



The Reality: A Multi-Core Processor



A Server with Multiple Processors

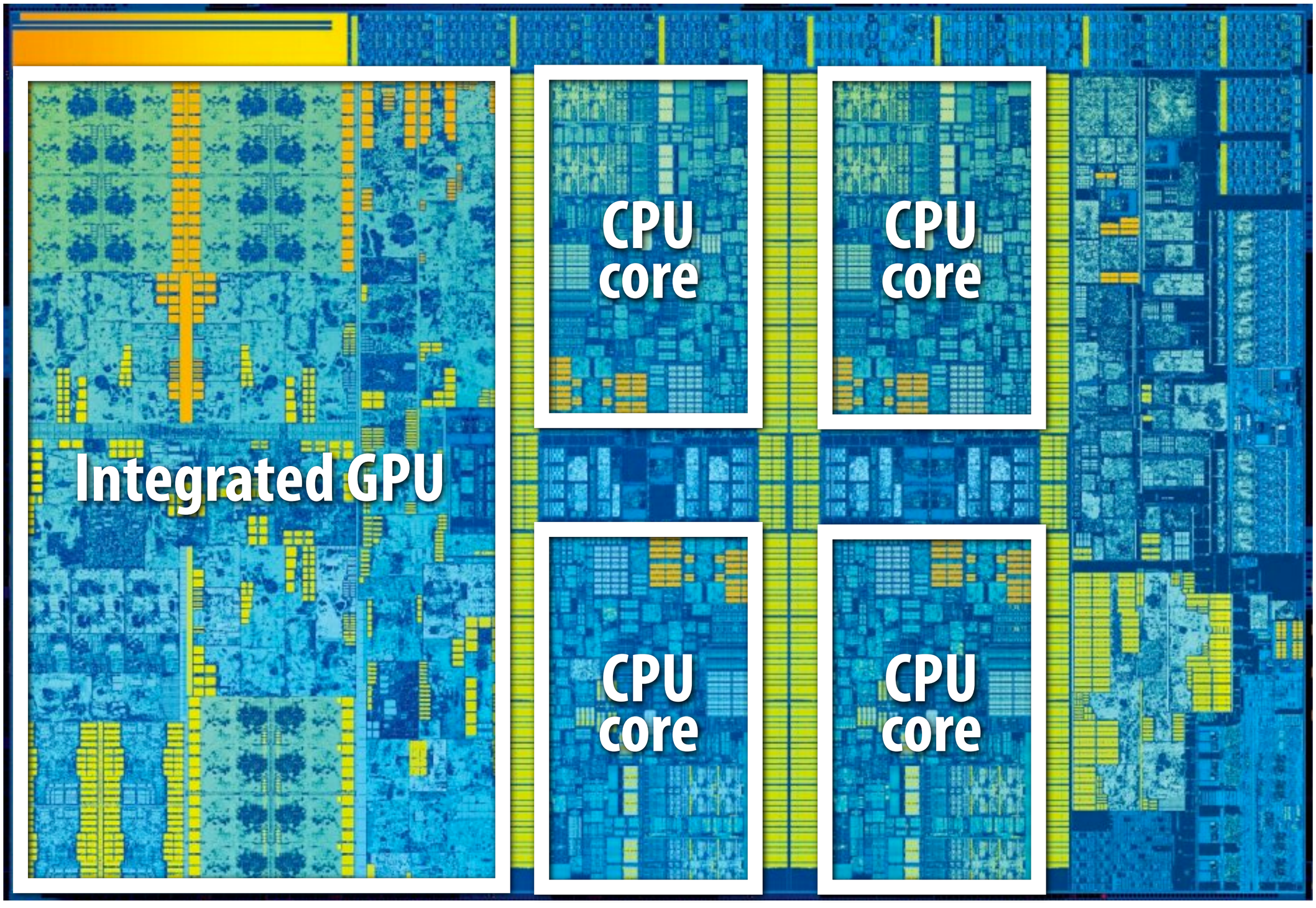


Motivation: Why Parallelism?

- The answer 15 years ago
  - To realize performance improvements that exceeded what CPU performance improvements could provide
    (specifically, in the early 2000's, what clock frequency scaling could provide)
  - Because if you just waited until next year, your code would run faster on the next generation CPU
- The answer today:
  - Because it is the primary way to achieve significantly higher application performance for the foreseeable future

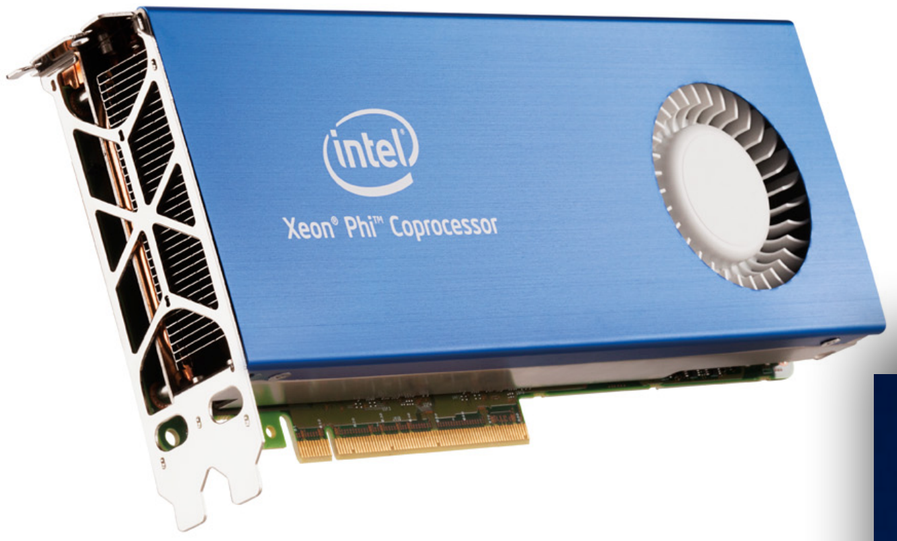
Intel Skylake (2015) (aka "6th generation Core i7")

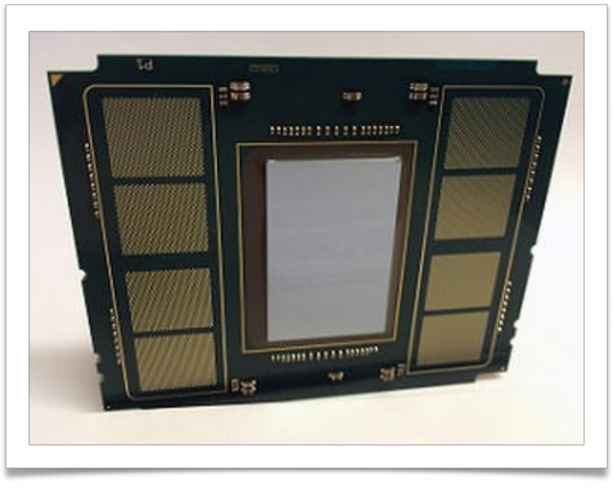
Quad-core CPU + multi-core GPU integrated on one chip



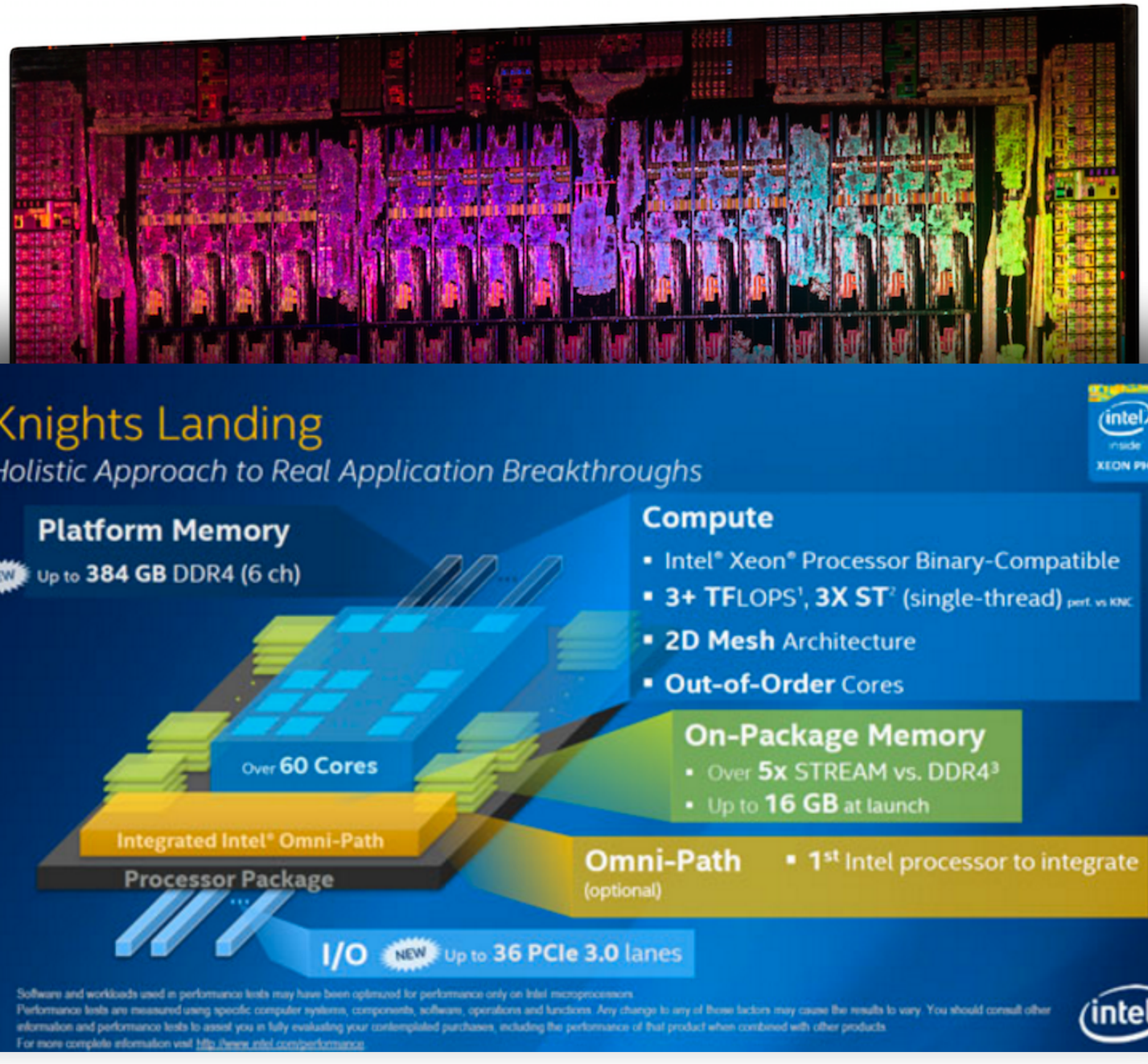
Intel Xeon Phi 7120A "coprocessor"

- MIC (Many Integrated Core) Architecture
- 61 "*simple*" x86 cores (1.3GHz, derived from Pentium)
- Targeted as an accelerator for supercomputing applications



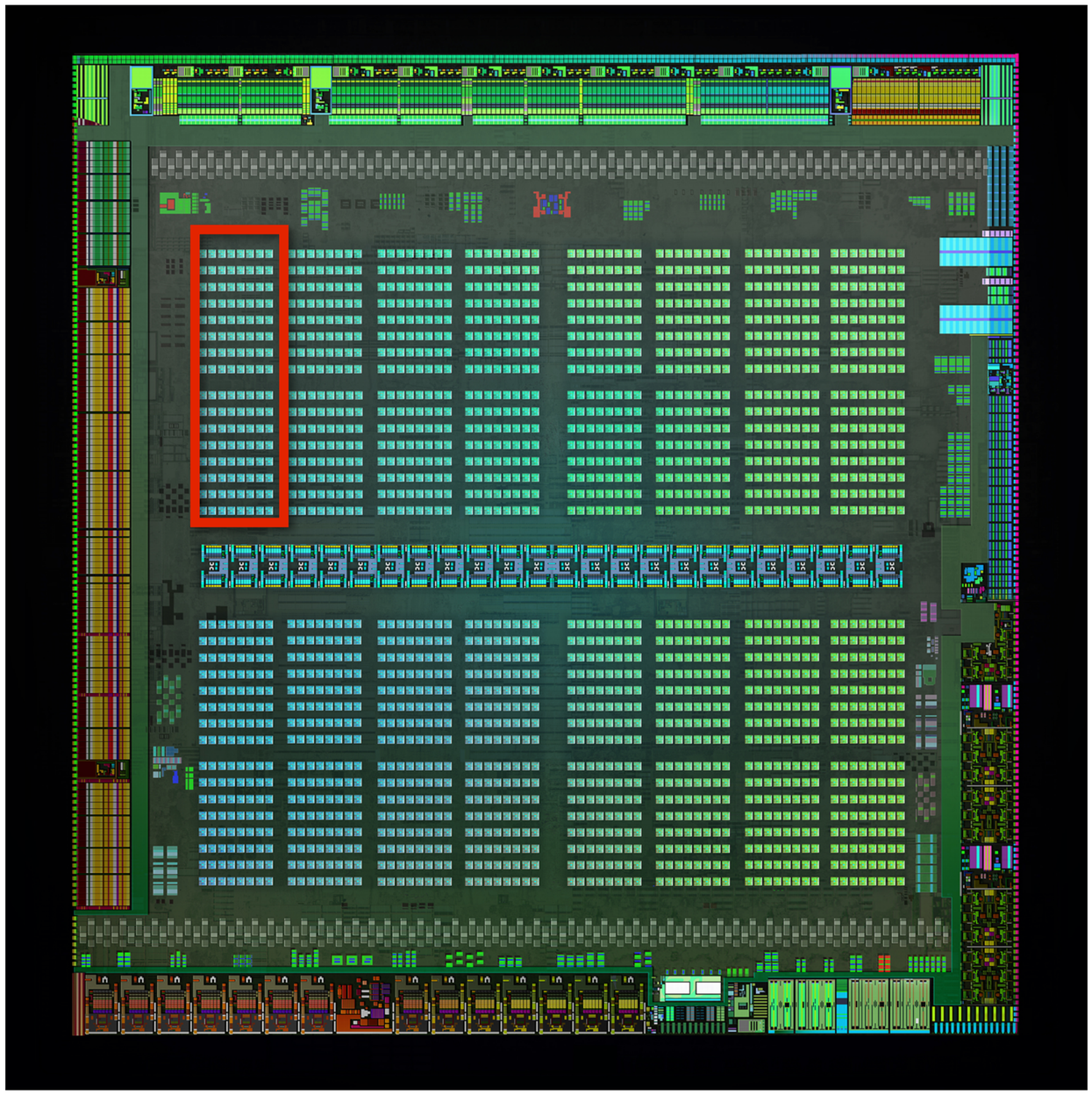


The Knights Landing Xeon Phi



NVIDIA Maxwell GTX 980 GPU (2014)

- Sixteen major processing blocks
  (but much, much more parallelism available…)

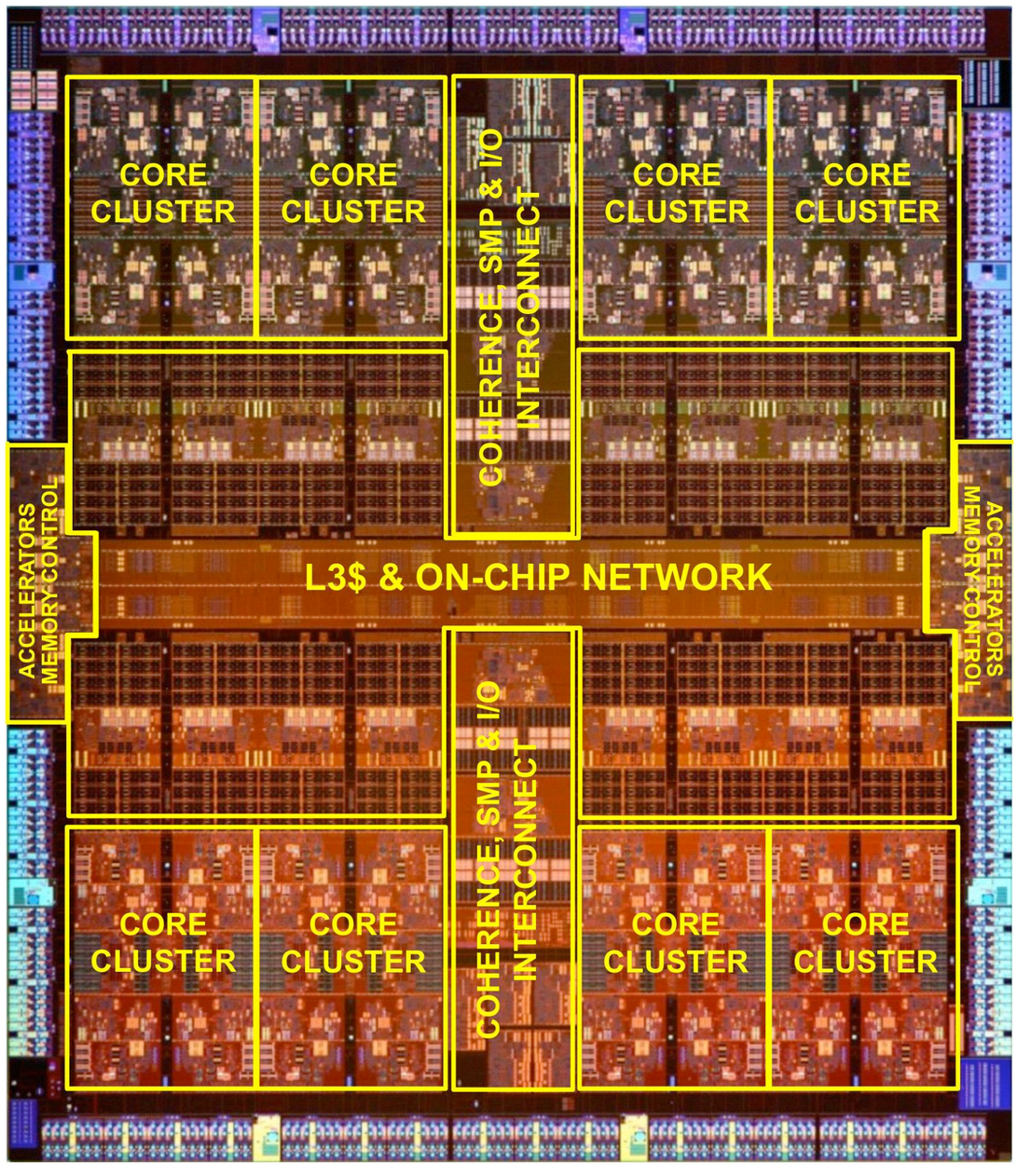


Supercomputing

- Today: clusters of multi-core CPUs + GPUs
- Oak Ridge National Laboratory: Titan (#2 supercomputer in the world)
  - 18,688 x 16 core AMD CPUs + 18,688 NVIDIA K20X GPUs

Software in Silicon

- **Oracle's SPARC M7 Processor: "SQL in Silicon"**
  - **accelerates processing streams of data from memory**
  - **Decompress, Scan, Select, Translate**



Oracle Announces Breakthrough Processor and Systems Design with SPARC M7

Dramatic Advancements in Memory Protection, Encryption Acceleration, and In-memory Database Processing Deliver End-to-End Security and Efficiency for Oracle Engineered Systems and Servers

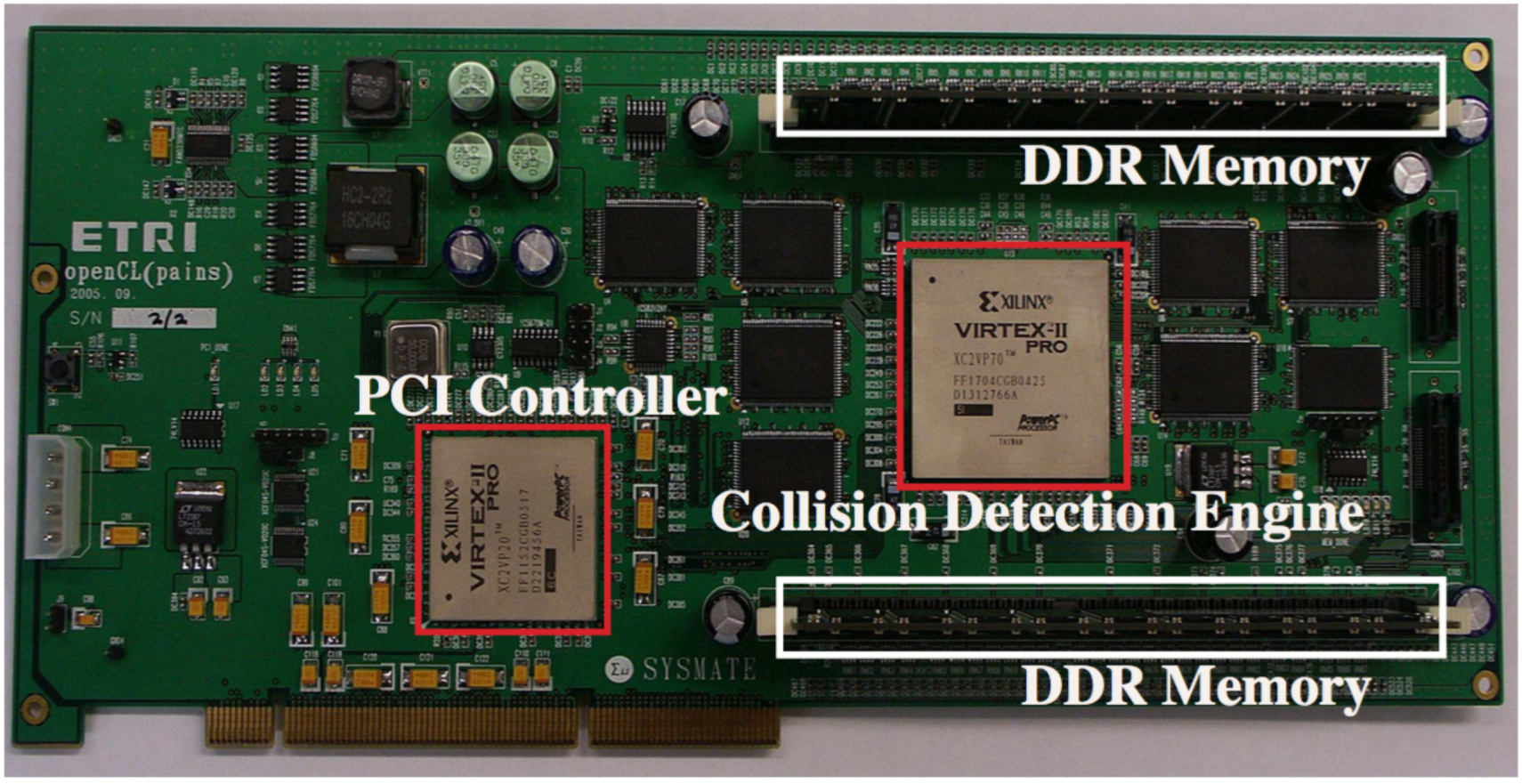
ORACLE OPENWORLD, SAN FRANCISCO —Oct 26, 2015

Oracle today introduced an all-new family of SPARC systems built on the revolutionary 32-core, 256-thread SPARC M7 microprocessor. The systems feature Security in Silicon for advanced intrusion protection and encryption; SQL in Silicon that delivers unparalleled database efficiency; and world record performance spanning enterprise, big data, and cloud applications.

The new SPARC M7 processor-based systems, including the Oracle SuperCluster M7 engineered system and SPARC T7 and M7 servers, are designed to seamlessly integrate with existing infrastructure and include fully integrated virtualization and management for cloud. All existing commercial and custom applications will run on SPARC M7 systems unchanged with significant improvements in security, efficiency, and performance. In addition, SPARC M7 is an open platform that developers can utilize to create new software that takes advantage of Security in Silicon and SQL in Silicon capabilities.

My Research Experience (2006)

- **Hardware-Accelerated Collision Detection**
  - FPGA-based prototype system
  - OpenCL [ETRI] : Open Collision Library



Hardware-Accelerated Ray-Triangle Intersection Testing for High-Performance Collision Detection

Sung-Soo Kim, Seung-Woo Nam, Do-Hyung Kim and In-Ho Lee
Digital Content Research Division
Electronics and Telecommunications Research Institute (ETRI), South Korea
{sungsoo, swnam, kdh99, leeinho}@etri.re.kr

ABSTRACT

We present a novel approach for hardware-accelerated collision detection. This paper describes the design of the hardware architecture for primitive inference testing components implemented on a multi-FPGA Xilinx Virtex-II prototyping system. This paper focuses on the acceleration of ray-triangle intersection operation which is the one of the most important operations in various applications such as collision detection and ray tracing. Also, the proposed hardware architecture is general for intersection operations of other object pairs such as sphere vs. sphere, oriented bounding box (OBB) vs. OBB, cylinder vs. cylinder and so on.

The result is a hardware-accelerated ray-triangle intersection engine that is capable of out-performing a 2.8GHz Xeon processor, running a well-known high performance software ray-triangle intersection algorithm, by up to a factor of seventy. In addition, we demonstrate that the proposed approach could prove to be faster than current GPU-based algorithms as well as CPU based algorithms for ray-triangle intersection.

Keywords: Collision Detection, Graphics Hardware, Intersection Testing, Ray Tracing.

1 INTRODUCTION

Collision detection is a fundamental task in many diverse applications, including surgical simulation, computer animation, computer games, robotics, physically-based simulation, automatic path finding, and virtual assembly simulation. The *collision query* checks whether two objects intersect and returns all pairs of overlapping features. We address the problem of collision query among collision primitives for interactive graphics applications. The set of collision primitives includes ray, axis-aligned bounding box (AABB), oriented bounding box (OBB), plane, cylinder, sphere and triangle.

The problem of fast and reliable collision detection has been extensively studied [Bergen04, Ericson04]. Despite the vast literature, real-time collision queries remain one of the major bottlenecks for interactive physically-based simulation and ray tracing. One of the challenges in the area is to develop the *custom hardware* for collision detection and ray tracing [ALB05, RBAZ05]. However, one major difficulty for implementing hardware is the multitude of collision detection and ray tracing algorithms. Dozens of algorithms and data structures exist for hierarchical scene traversal and intersection computation. Though the performance of these algorithms seems to be similar to software implementations, their applicability to hardware implementation has not yet been thoroughly investigated.

Since collision detection is such a fundamental task, it is highly desirable to have hardware acceleration available just like 3D graphics accelerators. Using specialized hardware, the system's CPU can be freed from computing collisions.

1.1 Main Results

We present a novel FPGA-accelerated architecture for fast collision detection among rigid bodies. Our proposed custom hardware for collision detection supports 13 intersection types among rigid bodies. In order to evaluate the proposed hardware architecture, we have performed our FPGA-accelerated implementation for accelerating intersection computations among collision primitives.

We demonstrate the effectiveness of our hardware architecture for collision queries in three scenarios: (a) ray-triangle intersection computation with 260 thousands of static triangles, (b) the same computation with dynamic triangles and (c) dynamic sphere-sphere intersection tesing. The performance of our FPGA-based hardware varies between 30 and 60 msec, depending on the complexity of the scene and the types of collision primitives. For our comparative study we also analyze three popular *ray-triangle intersection algorithms* to estimate on the size of hardware resource. More details are given in Section 4.

Journal of WSCG, ISSN 1213-6972, Vol. 15, 2007
WSCG'2007, January 29 – February 2, 2007
Plzen, Czech Republic.

HARDWARE-ACCELERATED RAY-TRIANGLE INTERSECTION TESTING FOR HIGH-PERFORMANCE COLLISION DETECTION

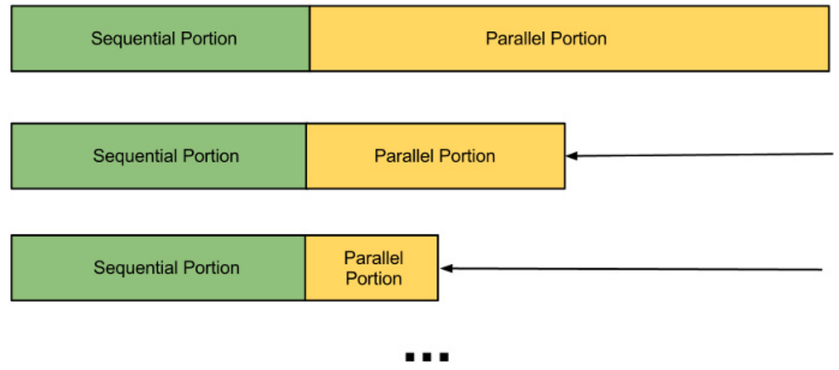
Journal of WSCG 2007

Observations

- Parallelism is inherent for today's computer architecture
- Degree of parallelism in future computer systems will increase
- Parallelism needs to be leveraged in all layers
  - Multi-Core
  - Main Memory - NUMA
  - Parallel Systems

Amdahl's Law (1967)

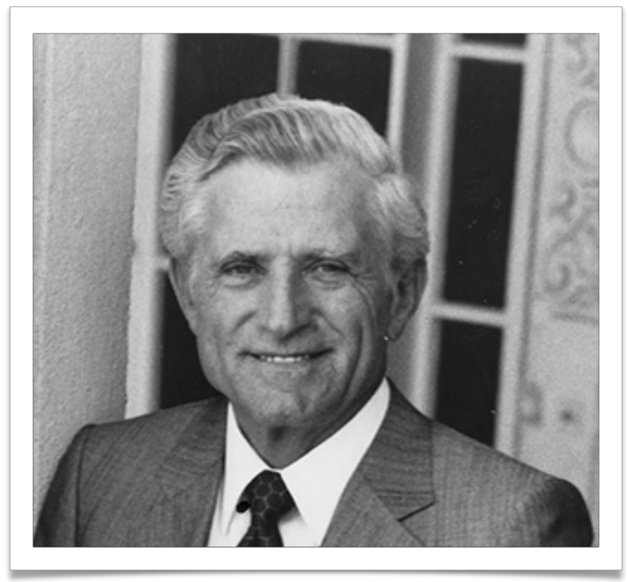
- How can we determine the effect of parallelizing an algorithm?
- The runtime is bound by *the longest sequential fraction* (*seq*) of the algorithm. For an unlimited number of processors, speed up can be calculated as:
  - Speedup = 1/*seq*, where *seq* = (1-*P*)



- To compute the speedup for a given number of processors, the following equation can be used:

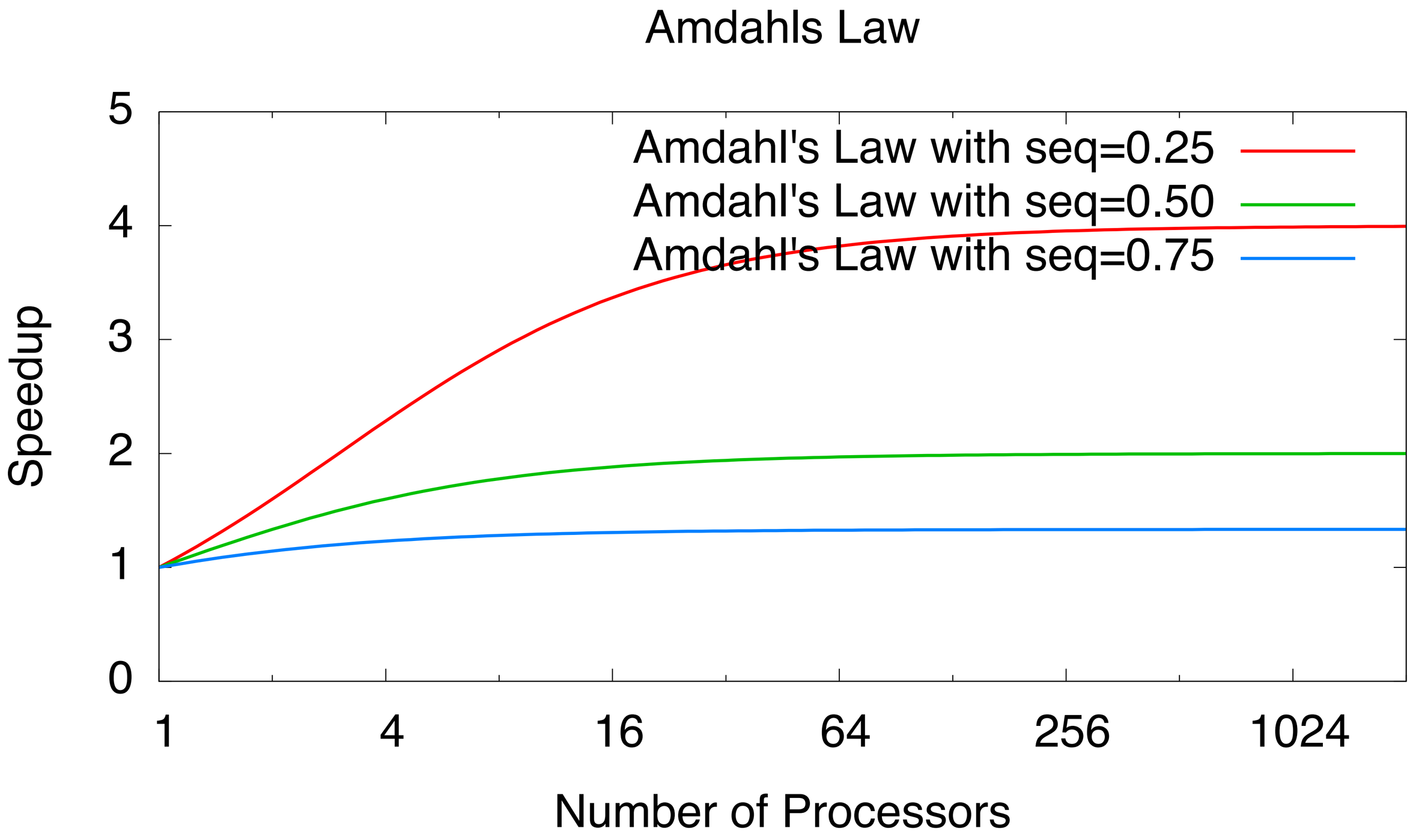
$$speedup = \frac{1}{S + \frac{P}{N}} = \frac{1}{(1-P) + \frac{P}{N}}$$

P: Parallel Fraction
S: Sequential Fraction
N: Number of processors



Gene Amdahl

Amdahl's Law

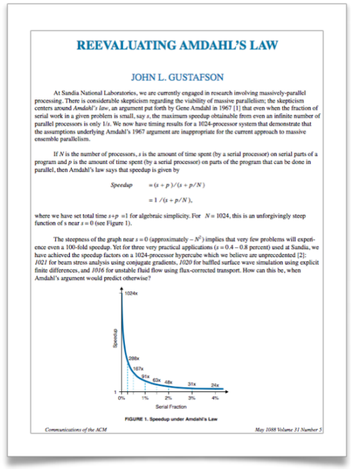

Amdahls Law
Amdahl's Law with seq=0.25
Amdahl's Law with seq=0.50
Amdahl's Law with seq=0.75
Speedup
0
1
2
3
4
5
1
4
16
64
256
1024
Number of Processors

Gustafson's Law (1988)

John L. Gustafson

"우리는 전산학에 몸담고 있는 사람들이 암달의 성능개선 공식을 잘못 사용함으로써 생기는 '정신적 한계점'을 넘어서야 한다고 생각한다.

성능 개선치 문제를 프로세서의 개수로 스케일링 하여 측정되어야 하며, 문제 크기를 고정시킨 상태에서 측정되면 안된다."



REEVALUATING AMDAHL'S LAW

JOHN L. GUSTAFSON

At Sandia National Laboratories, we are currently engaged in research involving massively-parallel processing. There is considerable skepticism regarding the viability of massive parallelism; the skepticism centers around *Amdahl's law*, an argument put forth by Gene Amdahl in 1967 [1] that even when the fraction of serial work in a given problem is small, say s, the maximum speedup obtainable from even an infinite number of parallel processors is only $1/s$. We now have timing results for a 1024-processor system that demonstrate that the assumptions underlying Amdahl's 1967 argument are inappropriate for the current approach to massive ensemble parallelism.

If N is the number of processors, s is the amount of time spent (by a serial processor) on serial parts of a program and p is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then Amdahl's law says that speedup is given by

$$Speedup \quad = (s + p)/(s + p/N)$$
$$= 1/(s + p/N),$$

where we have set total time $s + p = 1$ for algebraic simplicity. For $N = 1024$, this is an unforgivingly steep function of near $s = 0$ (see Figure 1).

The steepness of the graph near $s = 0$ (approximately $-N^2$) implies that very few problems will experience even a 100-fold speedup. Yet for three very practical applications ($s = 0.4 - 0.8$ percent) used at Sandia, we have achieved the speedup factors on a 1024-processor hypercube which we believe are unprecedented [2]: *1021* for beam stress analysis using conjugate gradients, *1020* for baffled surface wave simulation using explicit finite differences, and *1016* for unstable fluid flow using flux-corrected transport. How can this be, when Amdahl's argument would predict otherwise?

FIGURE 1. Speedup under Amdahl's Law

REEVALUATING AMDAHL'S LAW

Communications of the ACM 1988

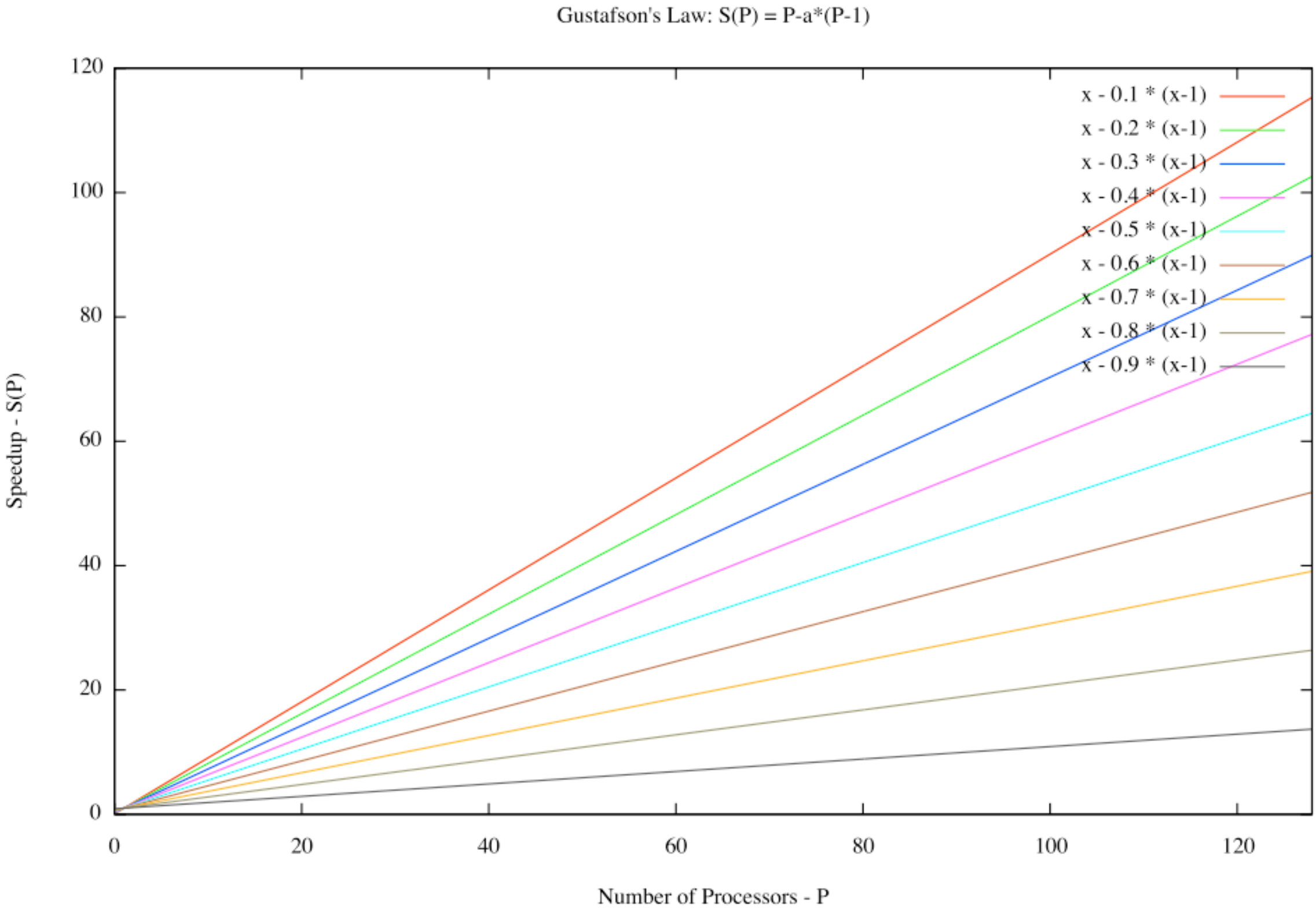
Gustafson's Law (1988)

- Gustafson and Barsis: People are typically **not** interested in the **shortest execution time**
  - Rather solve a bigger problem in reasonable time
- Problem size could then scale with the number of processors
  - Typical in *simulation* and *farmer / worker problems*
  - Leads to *larger parallel fraction* with increasing *N*
  - Serial part is usually *fixed* or *grows slower*
- Maximum scaled speedup (*S*) by *N* processors:

$$S = \frac{T_{SER} + N \cdot T_{PAR}}{T_{SER} + T_{PAR}} \qquad S = p - a(n) \cdot (1 - p)$$

- Linear speedup now becomes possible!
- Software needs to ensure that serial parts remain constant
- Other models exist (e.g. Work-Span model, Karp-Flatt metric)

Gustafson's Law (1988)



Parallel Programming Models

Techniques to program parallel hardware:

- Shared memory/threads
- Message passing
- Data parallelism
- Combinations (hybrid)

Shared Memory

- Concurrent tasks share common (logical) address space (does not imply physical shared memory)
- No explicit communication required
- Locks/semaphores are required to coordinate access
- Challenges:
  - *Data locality* is hard to manage (which data belongs to which process)
  - Scaling

Threads

- An OS process can start multiple threads
- Each thread has local data, but shares the resources of the parent OS process
- Threads communicate through global memory
- Threads are often associated with shared memory programming
- Require synchronization, e.g., locks/semaphores
- Example:
  - POSIX Threads
  - Win32 Threads
  - Intel Threading Building Block (TBB) Library

Message Passing

- Each concurrent task has own local memory
- Tasks can reside on one or on different machines
- State is kept and exchanged via message
- Tasks communicate and transfer data by sending/receiving messages
- Example:
  - OpenMPI
  - MPI/CH

Data Parallel

- Data resides in shared data structure (array, table, cube, …)
- Concurrent tasks work independently on data partitions
- All tasks perform the same operation
- Task scheduling is done by execution framework
- Example:
  - OpenMP "ParallelFor"
  - MapReduce

Parallelization with MapReduce

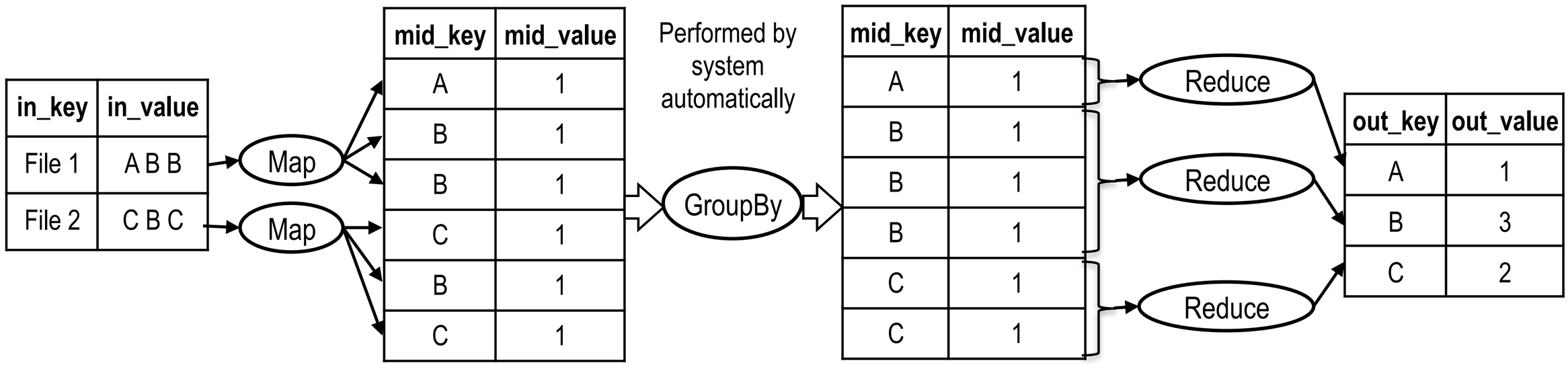
- MapReduce is a programming paradigm for shared-nothing cluster
  - Developed by Google/Yahoo to analyse large (petabyte) data-sets
  - Allows for *automatic parallelization* and *linear scaling* of MapReduce programs
- In a narrow sense, MapReduce describes a programming model based on a *map* and *reduce* function (both well known in *functional programming*)

  map(key1, value1) → list of <key2, value2>

  reduce(key2, list of <value2>) → list of <key3, value3>
  - *map* and *reduce* functions process <key, value> pairs independently and thus can be parallelized easily.
- In a wide sense, MapReduce describes an execution framework for distributing *map* and *reduce* tasks in a shared-nothing cluster

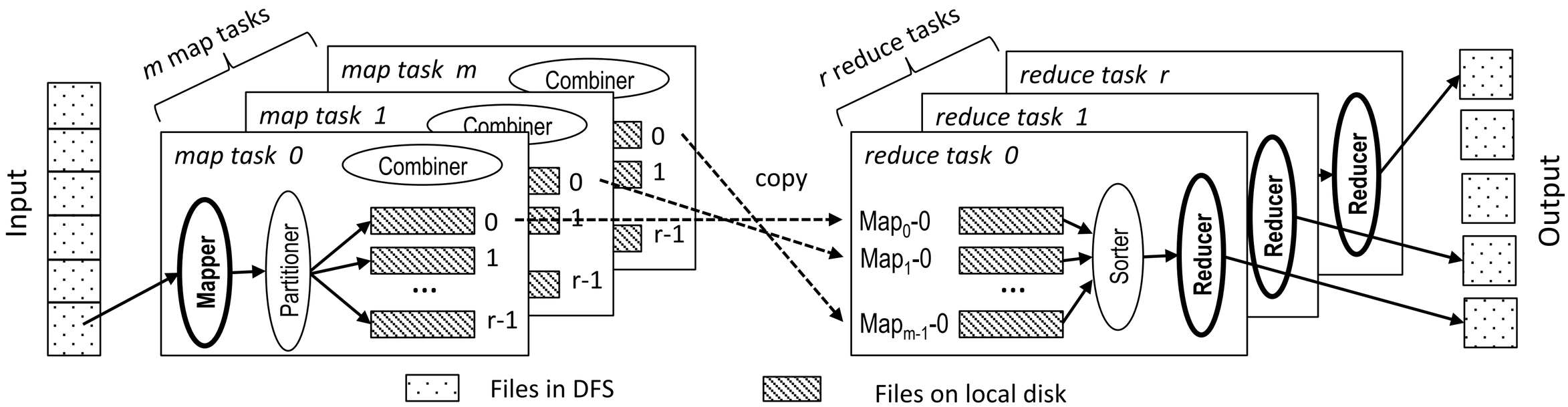
MapReduce Programming Model

Example: Word Count



1. **Input** is a list of key value pairs (e.g. keys="*file names*", values="*file content*")
2. A ***map function*** produces zero or more <*mid_key, mid_value*> pairs
3. The system performs a **group-by operation** on *mid_key*
4. A ***reduce function*** processes the list of *mid_values* belong to *mid_key* and produces a list of <out_key, out_value> pairs, here aggregate over values

MapReduce Execution Engine



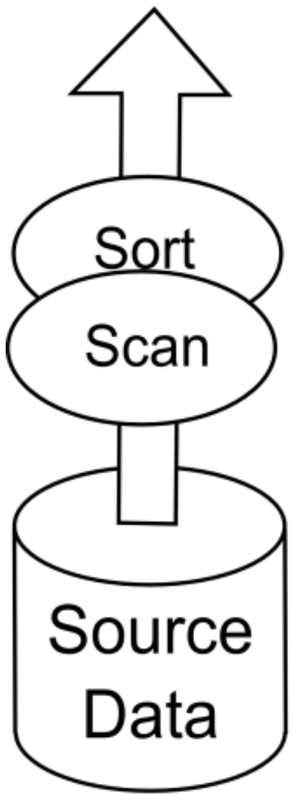
- Engine schedules map and reduce tasks, i.e. the execution of the map or reduce function on a <*key, value*> pair
- Programmer can focus on algorithm and has to implement only a map and reduce function
- The execution engine takes care of:
  - Task distribution: takes co-location of data into account
  - Shipping data between nodes: *communication* happens via disk!

Drawbacks and Problems

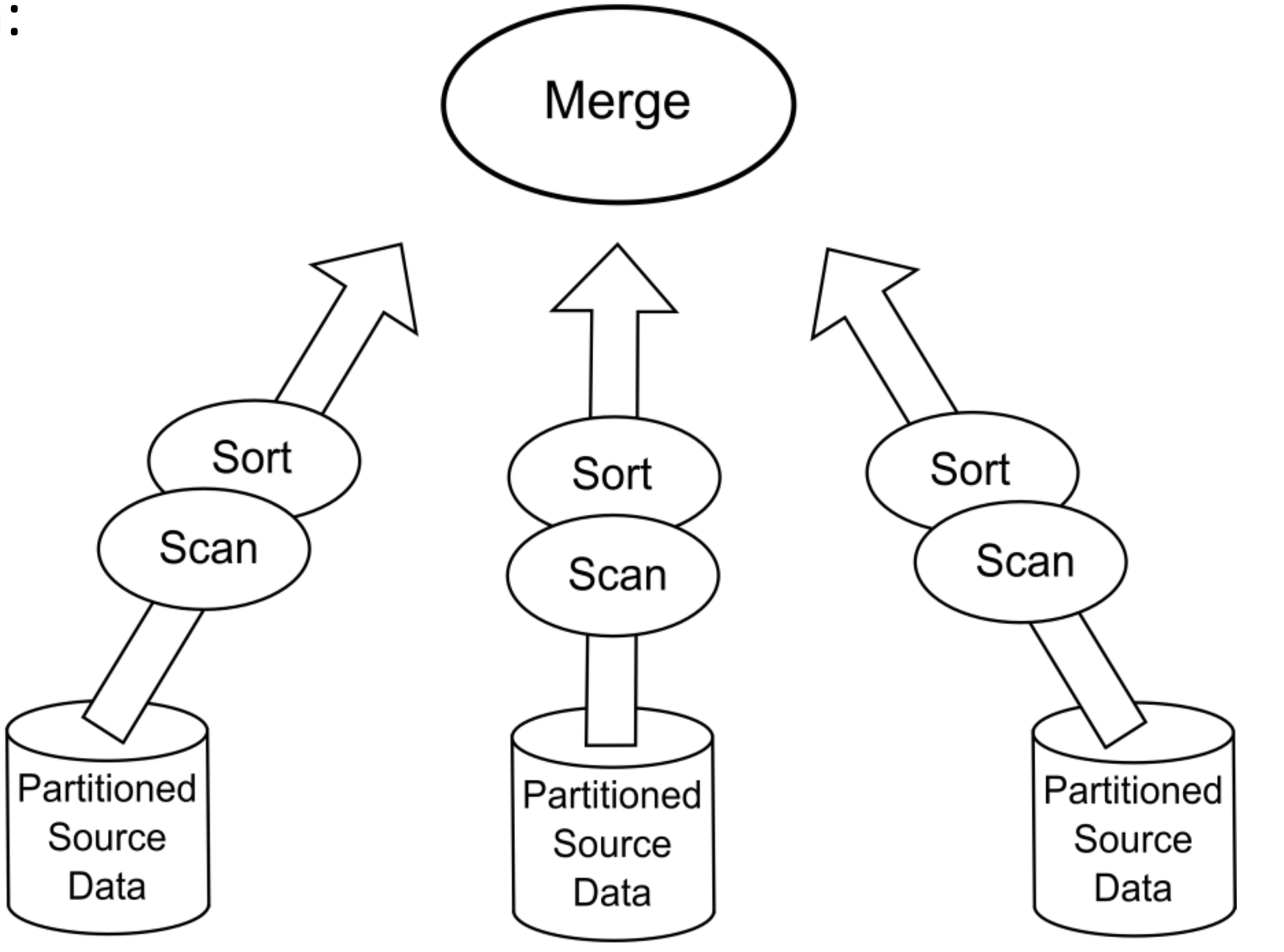
- Not all problems can be implemented efficiently with *map* and *reduce* functions (e.g. iterative algorithms)
- Batch-oriented execution model, thus not suited for interactive data analysis
- Communication via files is great for *fault-tolerance*, but inefficient for small data sets
- Good performance is only achieved if additional components e.g. *combiner, partitioner,* and *sorter* are also tuned by the programmer
  → Programmer cannot focus on algorithm only

Parallelism in In-Memory Databases

Types of Parallelism:

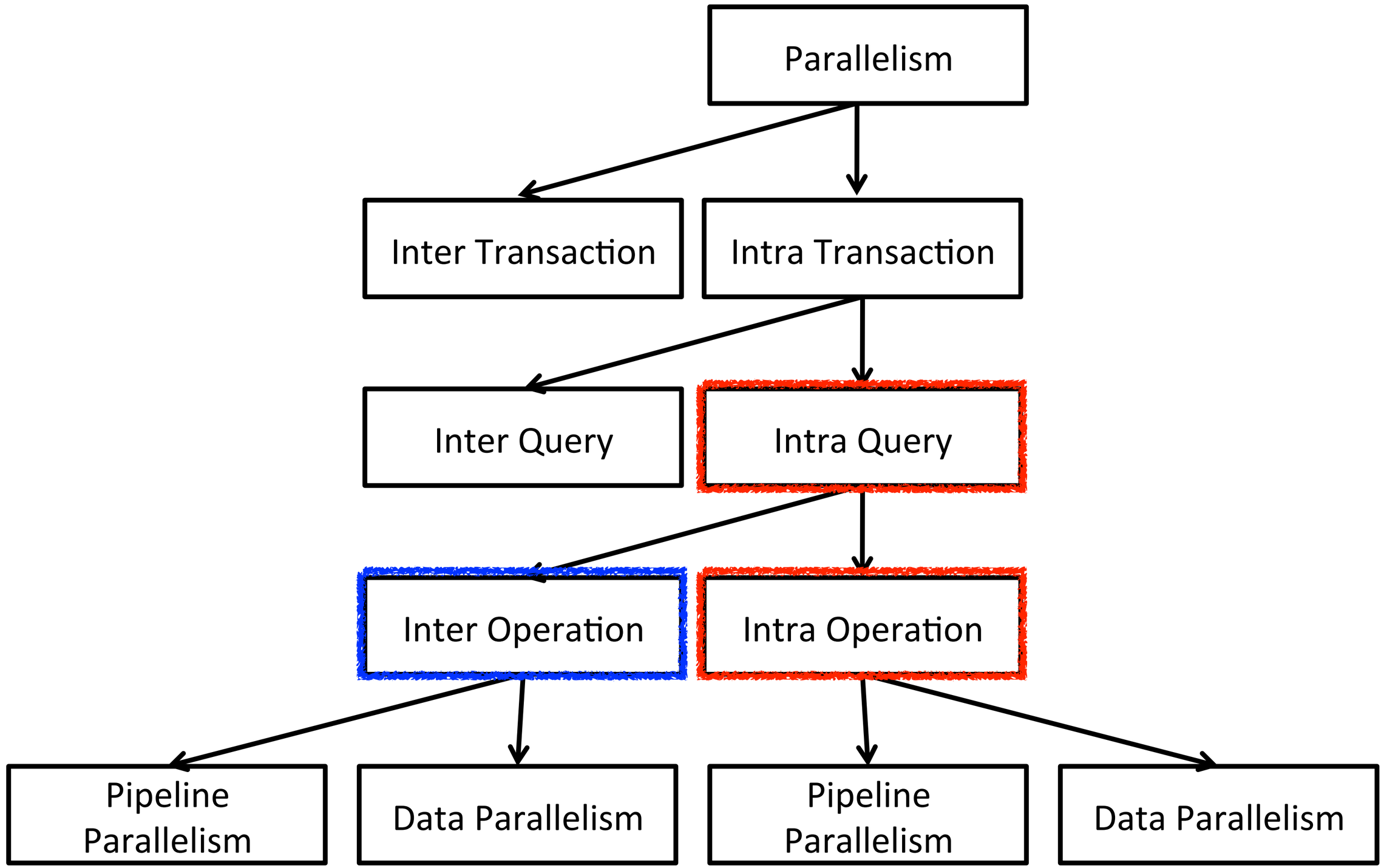


Pipeline Parallelism



Data Parallelism

Approaches to Parallelism

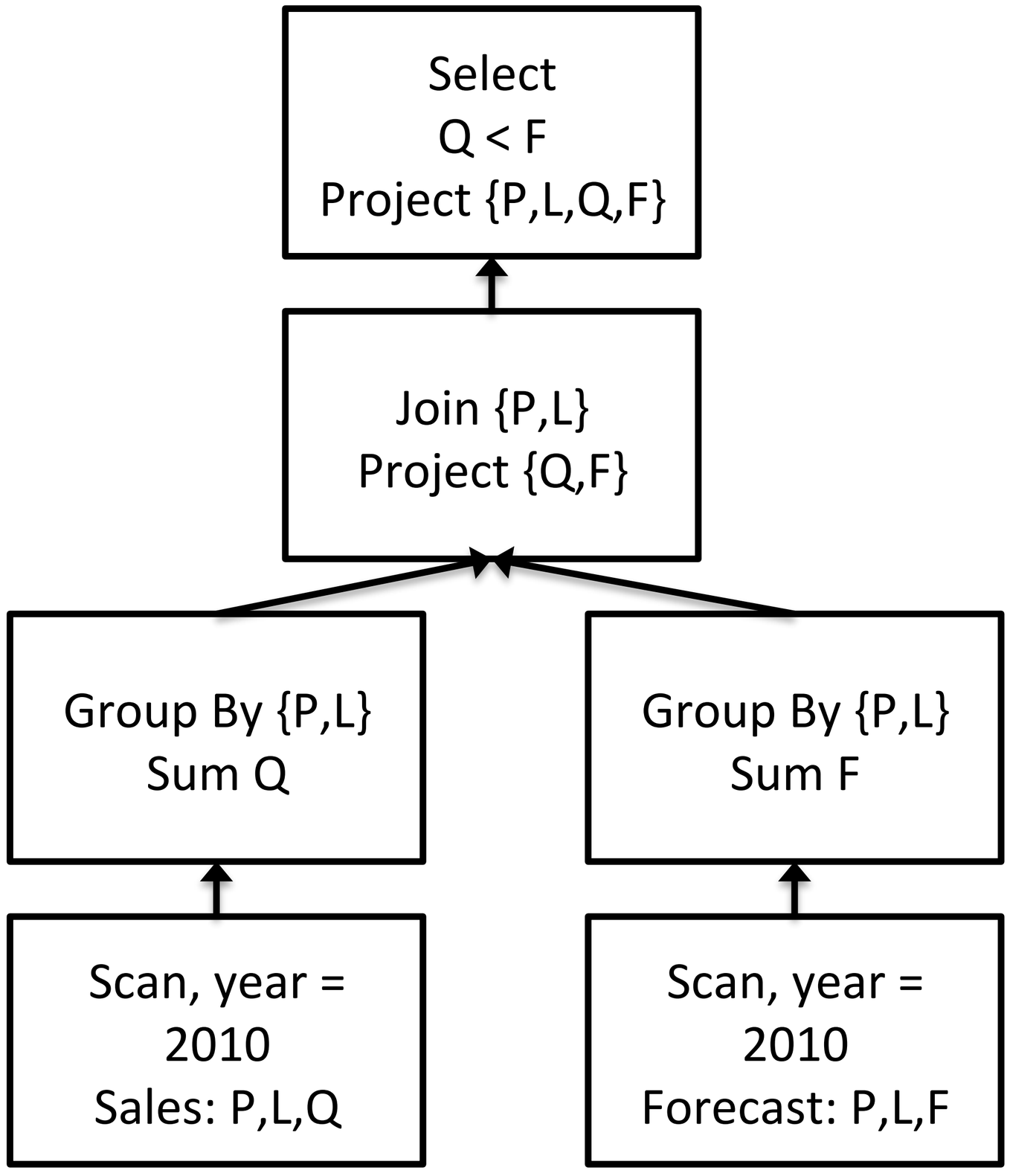


Example (Single Blade)

- Table "Sales"
  Columns: Product P, Location L, Quantity Q, Year Y
- Table "Forecast"
  Columns: Product P, Location L, Forecast F, Year Y
- Query:
  - "Which product at which location had lower quantity forecasted overall sales in 2010?"

```
SELECT Sales.P, Sales.L, SUM(Sales.Q) AS QTY, SUM(Forecast.F) AS FCST
FROM Sales, Forecast
WHERE Sales.P  = Forecast.P AND Sales.L = Forecast.L
    AND Sales.Y = 2010 AND Forecast.Y = 2010
GROUP BY Sales.P, Sales.L
HAVING QTY < FCST
```

Inter-Operator Parallelism (One Blade)



Example (Four Blades)

- Table "Sales"

  Columns: Product P, Location L, Quantity Q, Year Y

  Split into two parts: "Sales1" and "Sales2" based on P and L

  "Sales1" stored at "node1", "Sales2" stored at "node3"

- Table "Forecast"

  Columns: Product P, Location L, Forecast F, Year Y

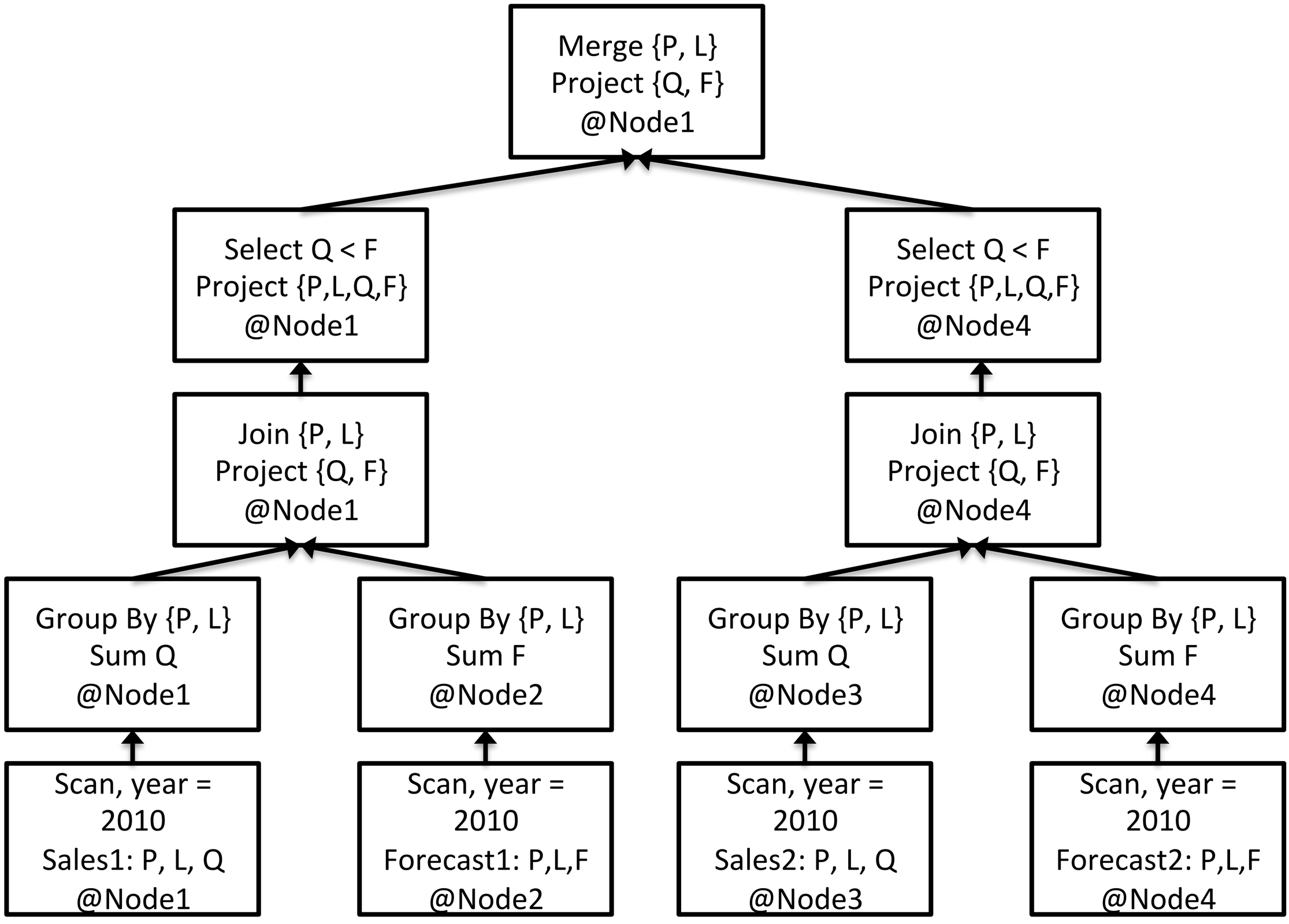
  Split into two parts: "Forecast1" and "Forecast2" based on P and L

  "Forecast1" stored at "node2", "Forecast2" stored at "node4"

- Query (remains the same, distribution *transparent* to the user):
  - "Which product at which location had lower quantity forecasted overall sales in 2010?"

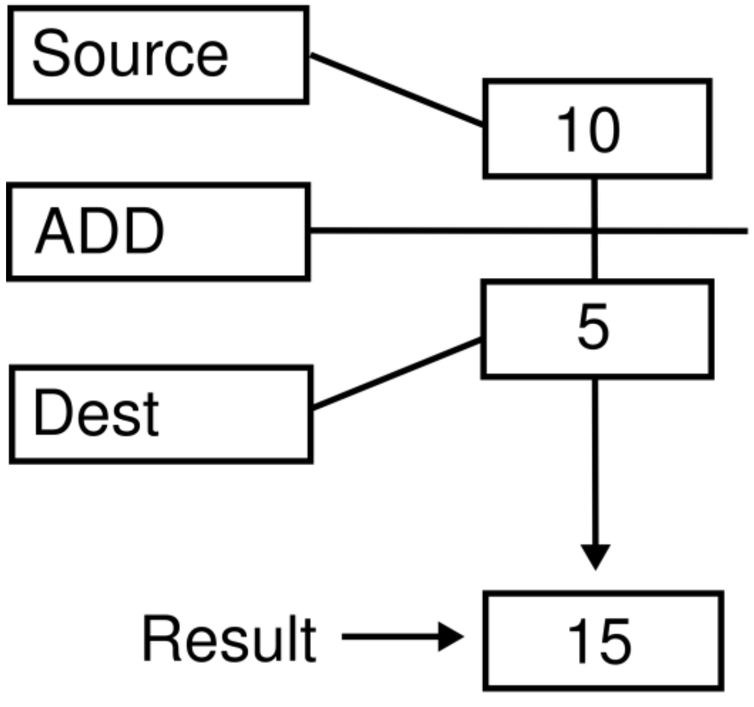
```
SELECT Sales.P, Sales.L, SUM(Sales.Q) AS QTY, SUM(Forecast.F) AS FCST
FROM Sales, Forecast
WHERE Sales.P  = Forecast.P AND Sales.L = Forecast.L
    AND Sales.Y = 2010 AND Forecast.Y = 2010
GROUP BY Sales.P, Sales.L
HAVING QTY < FCST
```

Inter-Operator Parallelism (Four Blades)

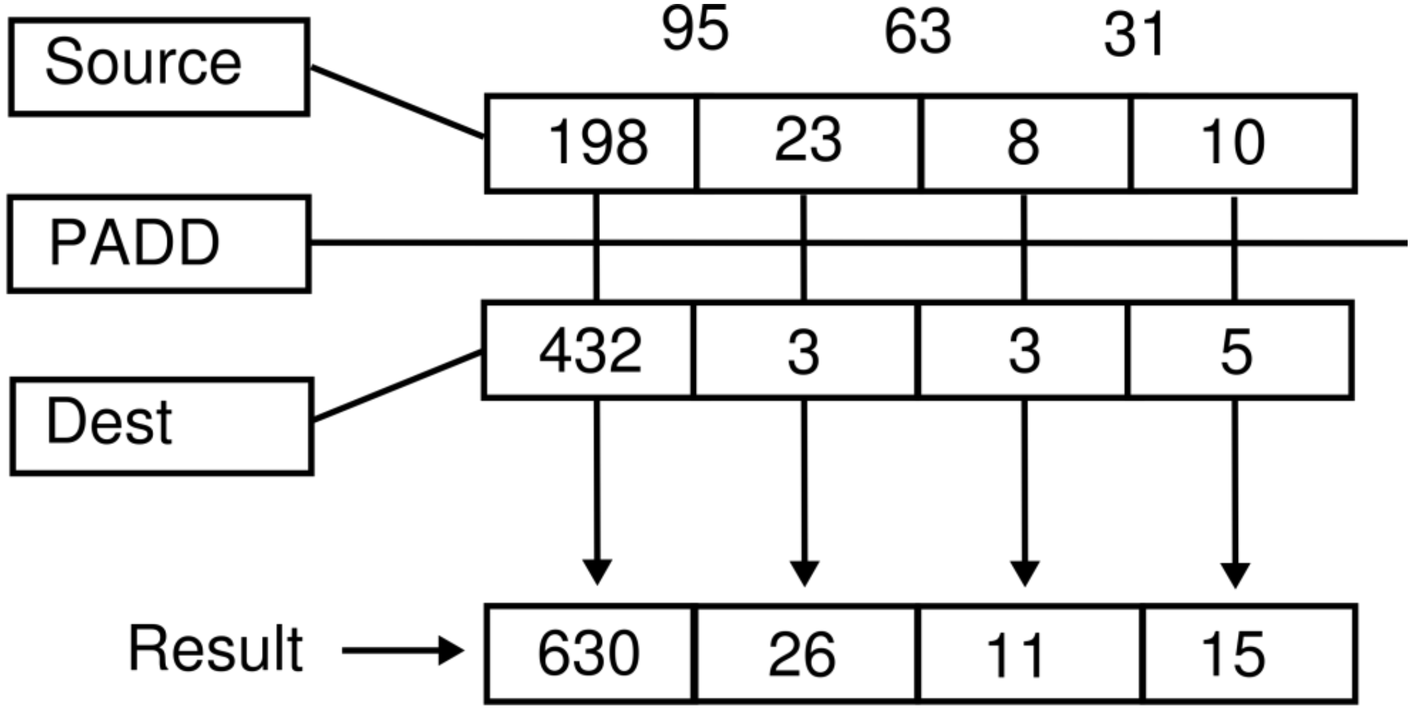


SIMD Multiple Data Operations per Cycle

- A class of CPU instructions that allow the processor to perform the same operation on multiple data points simultaneously.
- Both current AMD and Intel CPUs have ISA and microarchitecture support SIMD operations.
  - MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX



Scalar Add



Parallel Add
(Low-Level Parallel Column Access)

Key Observations

- Algorithm
  - 2-stage pipeline (pipeline parallelism)
  - Programming Model: shared memory/threads, data parallel
- Synchronization: every thread writes into its private data structure
  → no synchronization required for that
- Data skew handled by *small* input work package size (compared to fact table size)
- *Cache-awareness* due to fixed size of local hash tables
- Main-memory consumption bounded by buffer size
- Number of threads can be adjusted
- *Smaller* algorithm for parallel join computation

Inter-Query Parallelism

- Improve overall performance by allowing multiple queries to execute simultaneously.
  - Provide the illusion of *isolation* through *concurrency control scheme*.

Intra-Query Parallelism

- Improve the performance of a single query by *executing its operators in parallel*.

- Approach #1: Intra-Operator (Horizontal)
  - Operators are decomposed into independent instances that perform the same function on different subsets of data.

- Approach #2: Inter-Operator (Vertical)
  - Operations are overlapped in order to pipeline data from one stage to the next without materialization.

Volcano: Intra-Operator Parallelism

120 IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 6, NO. 1, FEBRUARY 1994

Volcano—An Extensible and Parallel Query Evaluation System

Goetz Graefe

Abstract—**To investigate the interactions of extensibility and parallelism in database query processing, we have developed a new dataflow query execution system called Volcano. The Volcano effort provides a rich environment for research and education in database systems design, heuristics for query optimization, parallel query execution, and resource allocation.**

Volcano uses a standard interface between algebra operators, allowing easy addition of new operators and operator implementations. Operations on individual items, e.g., predicates, are imported into the query processing operators using *support functions*. The semantics of support functions is not prescribed; any data type including complex objects and any operation can be realized. Thus, Volcano is *extensible* with new operators, algorithms, data types, and type-specific methods.

Volcano includes two novel *meta-operators*. The *choose-plan* meta-operator supports *dynamic query evaluation plans* that allow delaying selected optimization decisions until run-time, e.g., for embedded queries with free variables. The *exchange* meta-operator supports *intra-operator parallelism* on partitioned datasets and both *vertical and horizontal inter-operator parallelism*, translating between demand-driven dataflow within processes and data-driven dataflow between processes.

All operators, with the exception of the exchange operator, have been designed and implemented in a single-process environment, and parallelized using the exchange operator. Even operators not yet designed can be parallelized using this new operator if they use and provide the interator interface. Thus, the issues of data manipulation and parallelism have become *orthogonal*, making Volcano the first implemented query execution engine that effectively combines extensibility and parallelism.

Index Terms—**Dynamic query evaluation plans, extensible database systems, iterators, operator model of parallelization, query execution.**

I. Introduction

IN ORDER to investigate the interactions of extensibility, efficiency, and parallelism in database query processing and to provide a testbed for databse systems research and education, we have designed and implemented a new query evaluation system called Volcano. It is intended to provide an experimental vehicle for research into query execution techniques and query optimization optimization heuristics rather than a database system ready to support applications. It is not a complete database system as it lacks features such as a user-friendly query language, a type system for instances (record definitions), a query optimizer, and catalogs. Because of this focus, Volcano is able to serve as an experimental vehicle for a multitude of purposes, all of them open-ended, which results in a combination of requirements that have not been integrated in a single system before. First, it is modular and extensible to enable future research, e.g., on algorithms, data models, resource allocation, parallel execution, load balancing, and query optimization heuristics. Thus, Volcano provides an infrastructure for experimental research rather than a final research prototype in itself. Second, it is simple in its design to allow student use and research. Modularity and simplicity are very important for this purpose because they allow students to begin working on projects without an understanding of the entire design and all its details, and they permit several concurrent student projects. Third, Volcano's design does not presume any particular data model; the only assumption is that query processing is based on transforming sets of items using parameterized operators. To achieve data model independence, the design very consistently separates set processing *control* (which is provided and inherent in the Volcano operators) from *interpretation and manipulation* of data items (which is imported into the operators, as described later). Fourth, to free algorithm design, implementation, debugging, tuning, and initial experimentation from the intricacies of parallelism but to allow experimentation with parallel query processing. Volcano can be used as a single-process or as a parallel system. Single-process query evaluation plans can already be parallelized easily on shared-memory machines and soon also on distributed-memory machines. Fifth, Volcano is realistic in its query execution paradigm to ensure that students learn how query processing is really done in commercial database products. For example, using temporary files to transfer data from one operation to the next as suggested in most textbooks has a substantial performance penalty, and is therefore used in neither real database systems nor in Volcano. Finally, Volcano's means for parallel query processing could not be based on existing models since all models explored to date have been designed with a particular data model and operator set in mind. Instead, our design goal was to make parallelism and data manipulation *orthogonal*, which means that the mechanisms for parallel query processing are independent of the operator set and semantics, and that all operators, including new

Manuscript received July 26, 1990; revised September 5, 1991. This work was supported in part by the National Science Foundation under Grants IRI-8996270, IRI-8912618, and IRI-9006348, and by the Oregon Advanced Computing Institute (OACIS).

The author is with the Computer Science Department, Portland State University, Portland, OR 97207-0751.

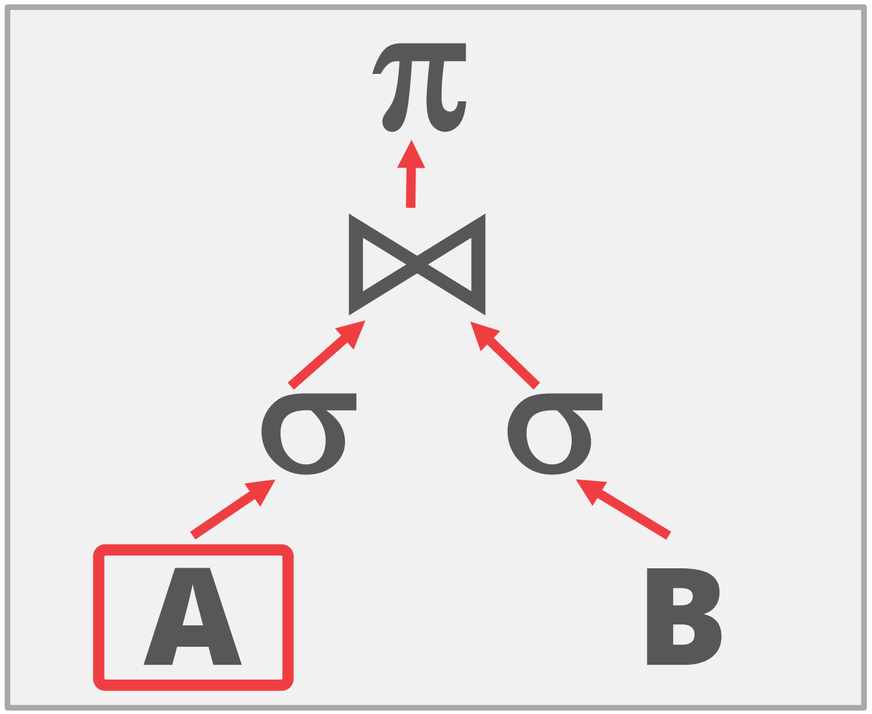
IEEE Log Number 9211308.

VOLCANO-AN EXTENSIBLE AND PARALLEL QUERY EVALUATION SYSTEM

IEEE TKDE 1994

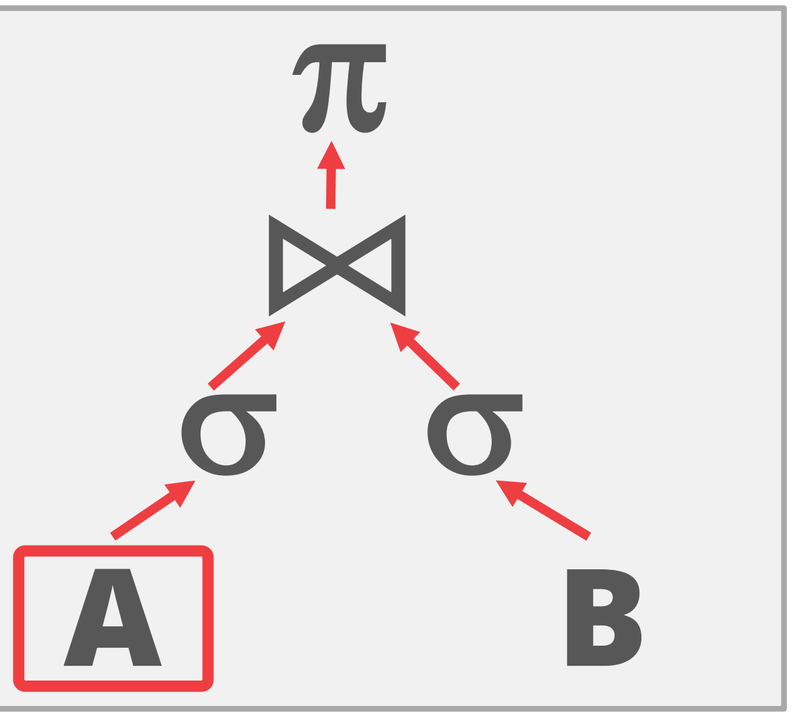
Intra-Operator Parallelism


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



Intra-Operator Parallelism

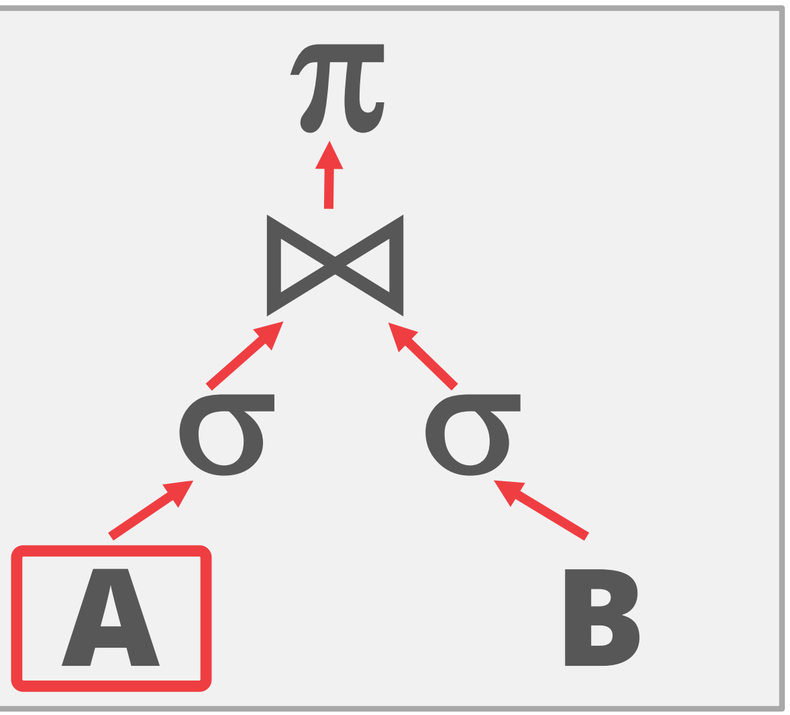

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

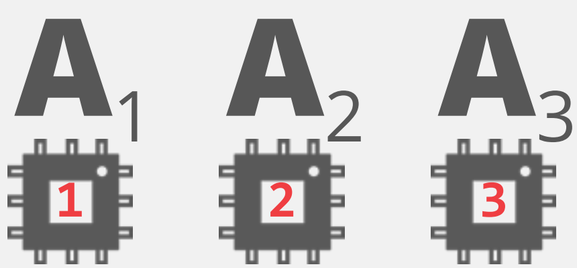


A_1 A_2 A_3

Intra-Operator Parallelism

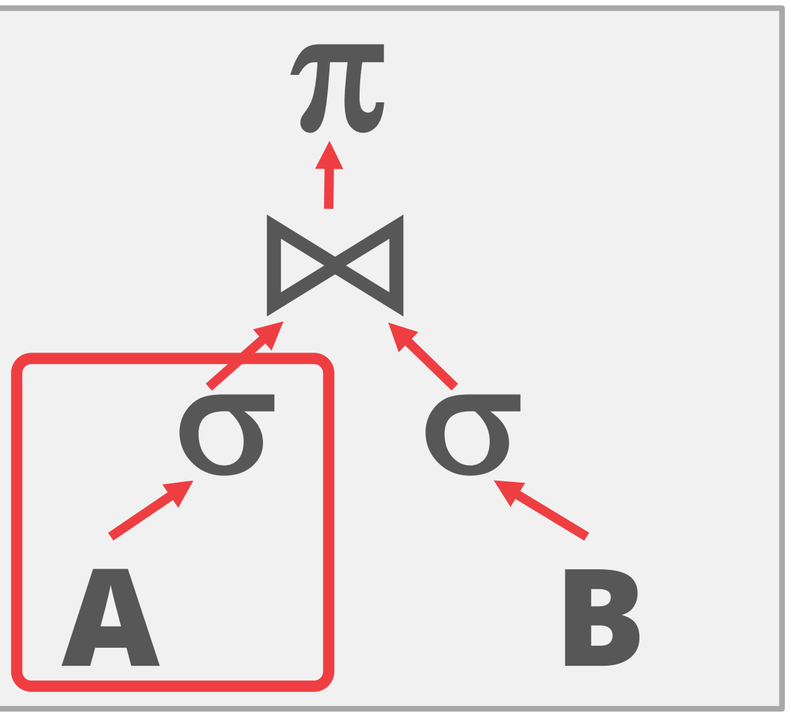

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

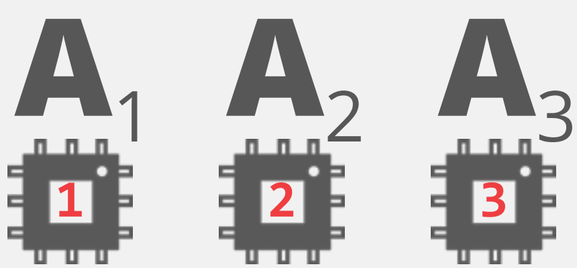




Intra-Operator Parallelism

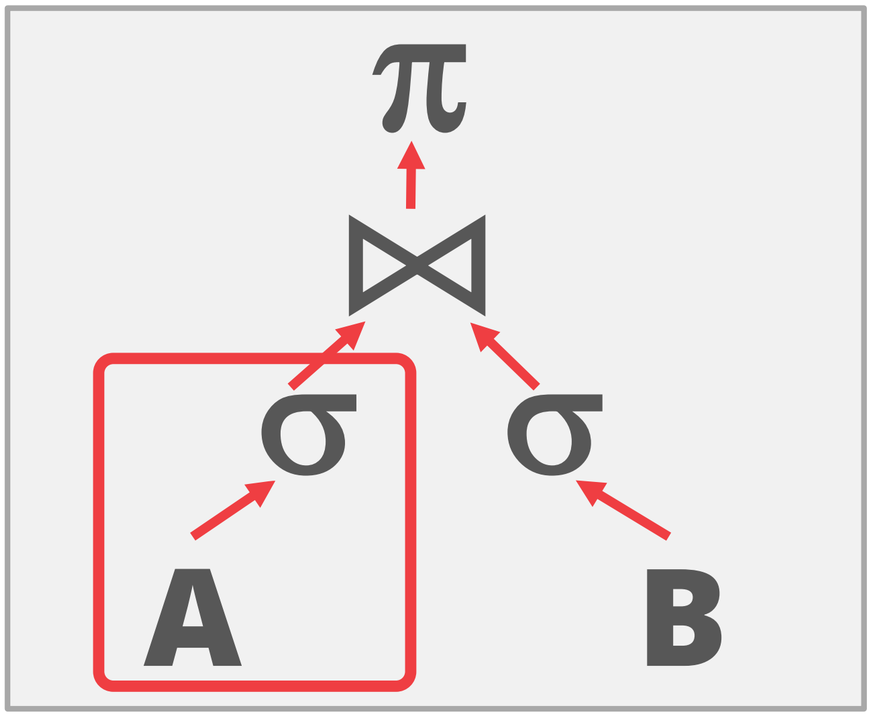

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

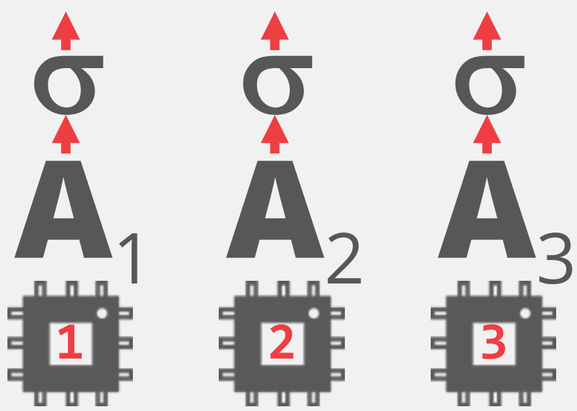




Intra-Operator Parallelism

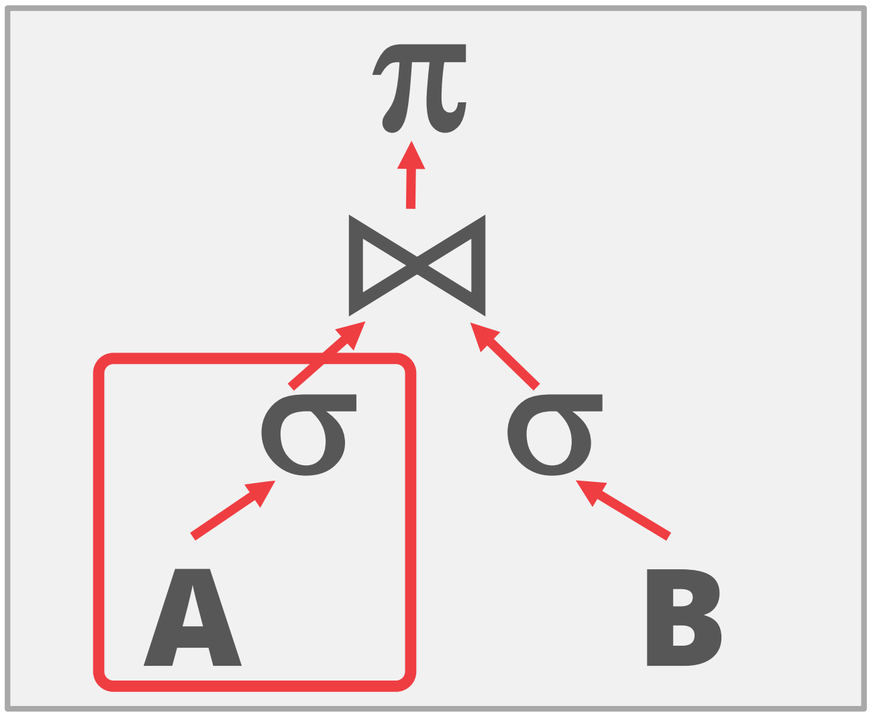

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

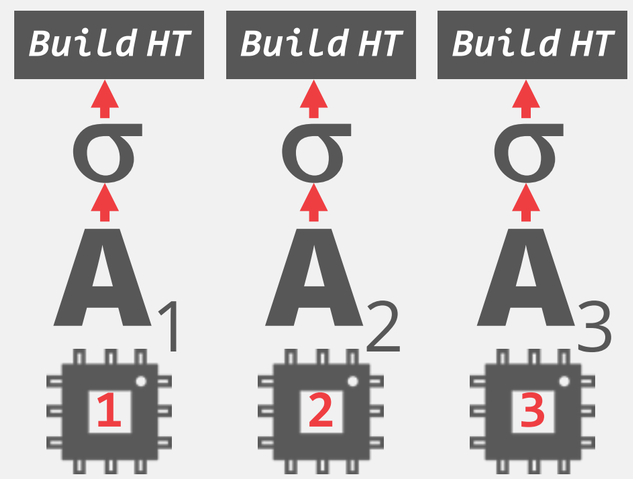




Intra-Operator Parallelism

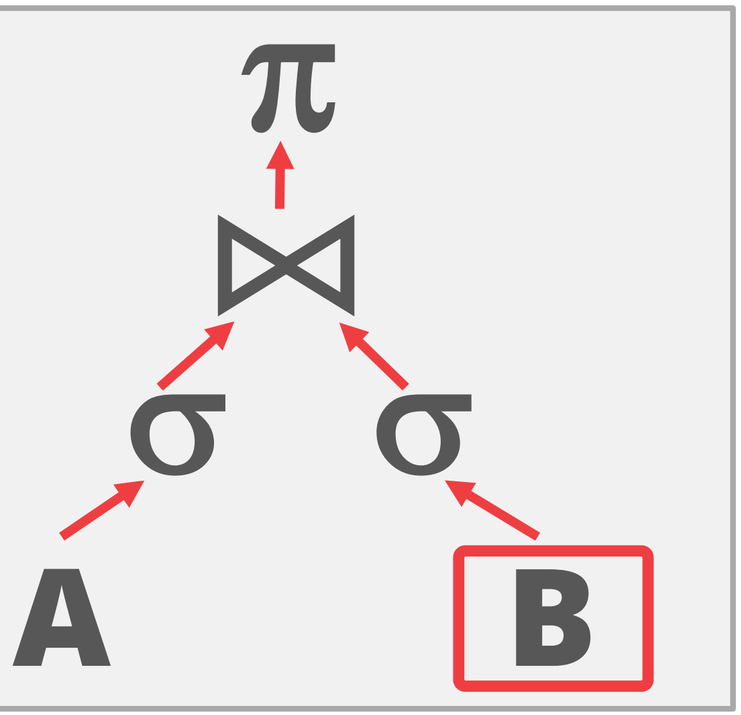

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

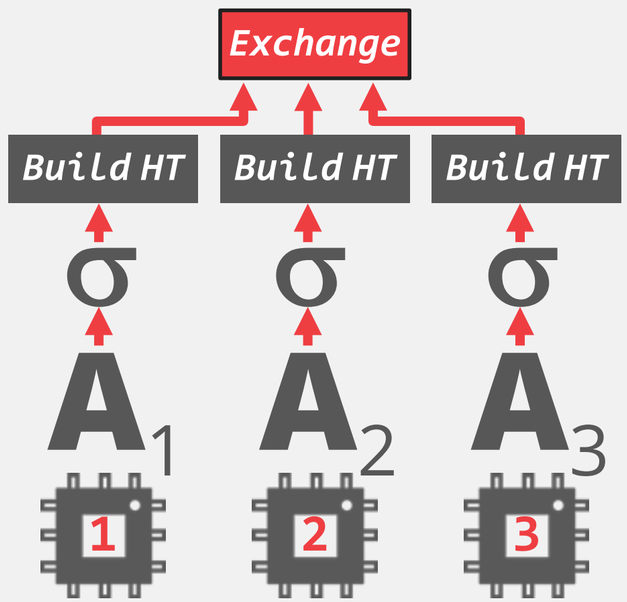




Intra-Operator Parallelism

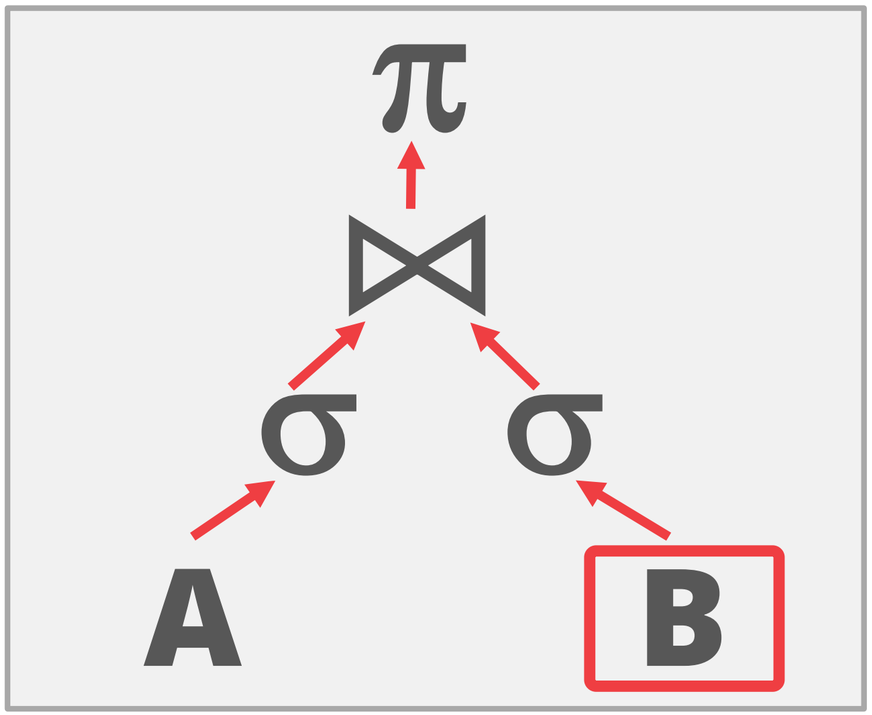

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

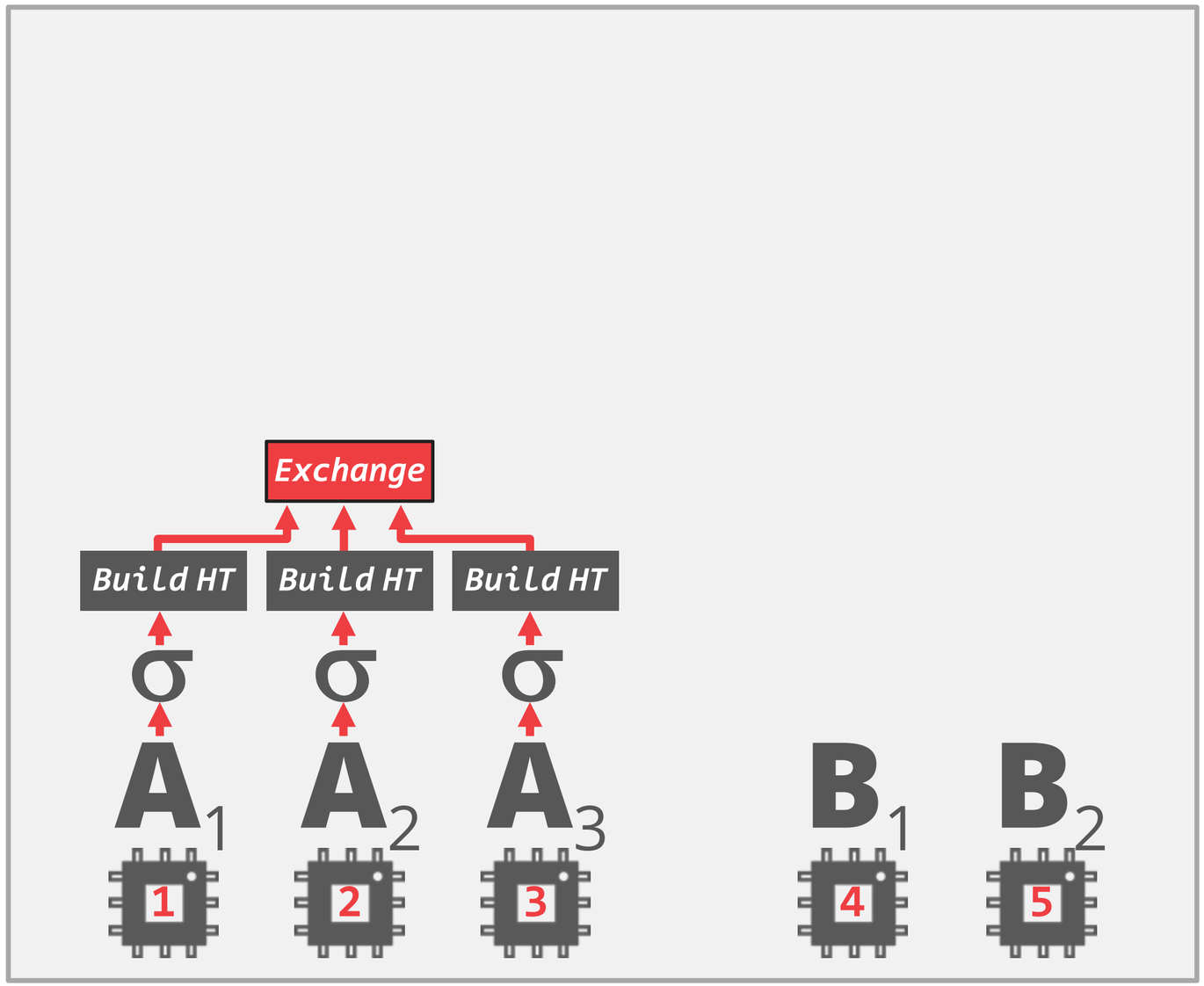




Intra-Operator Parallelism

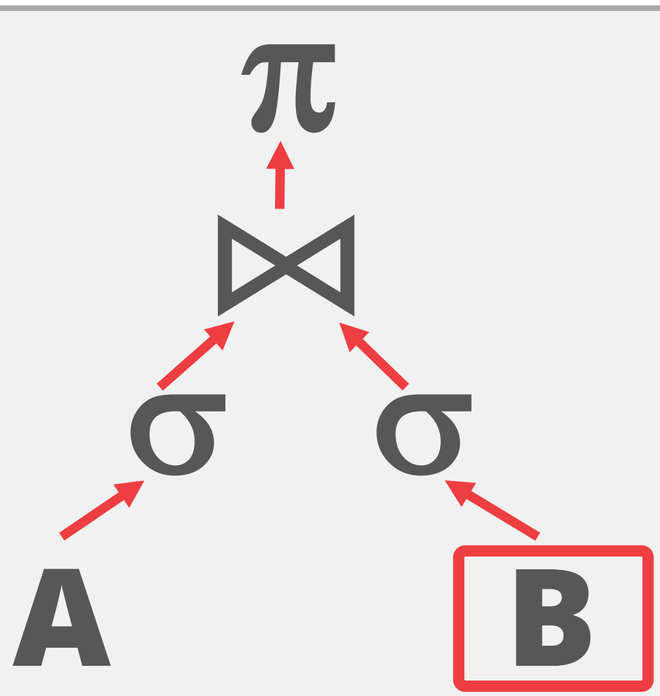

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

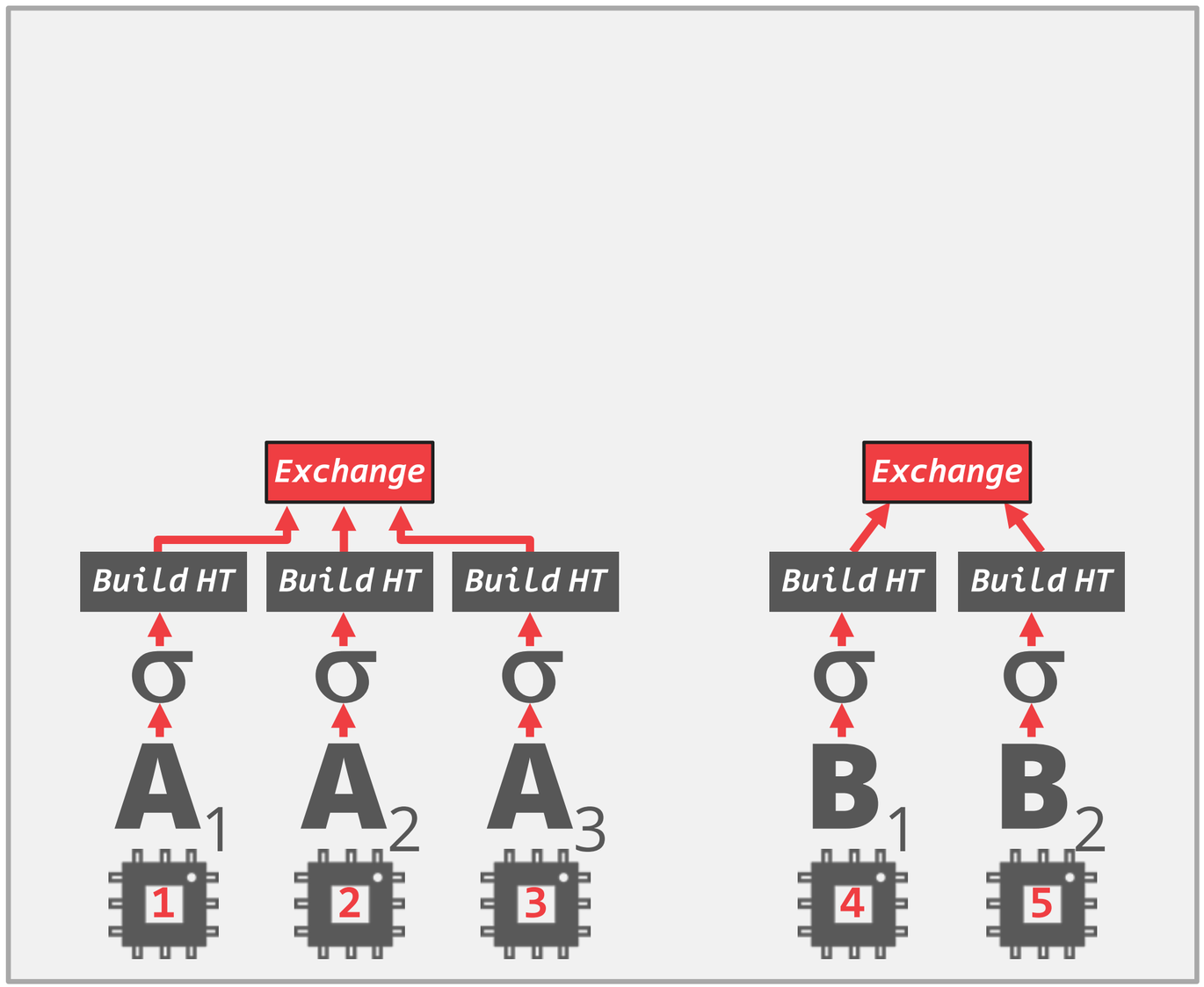




Intra-Operator Parallelism

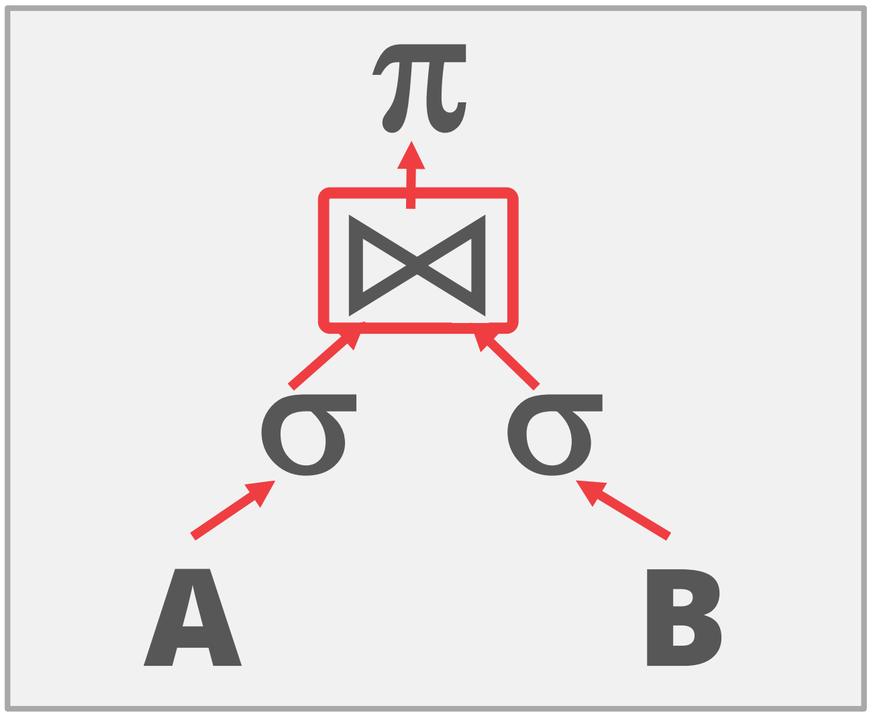

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

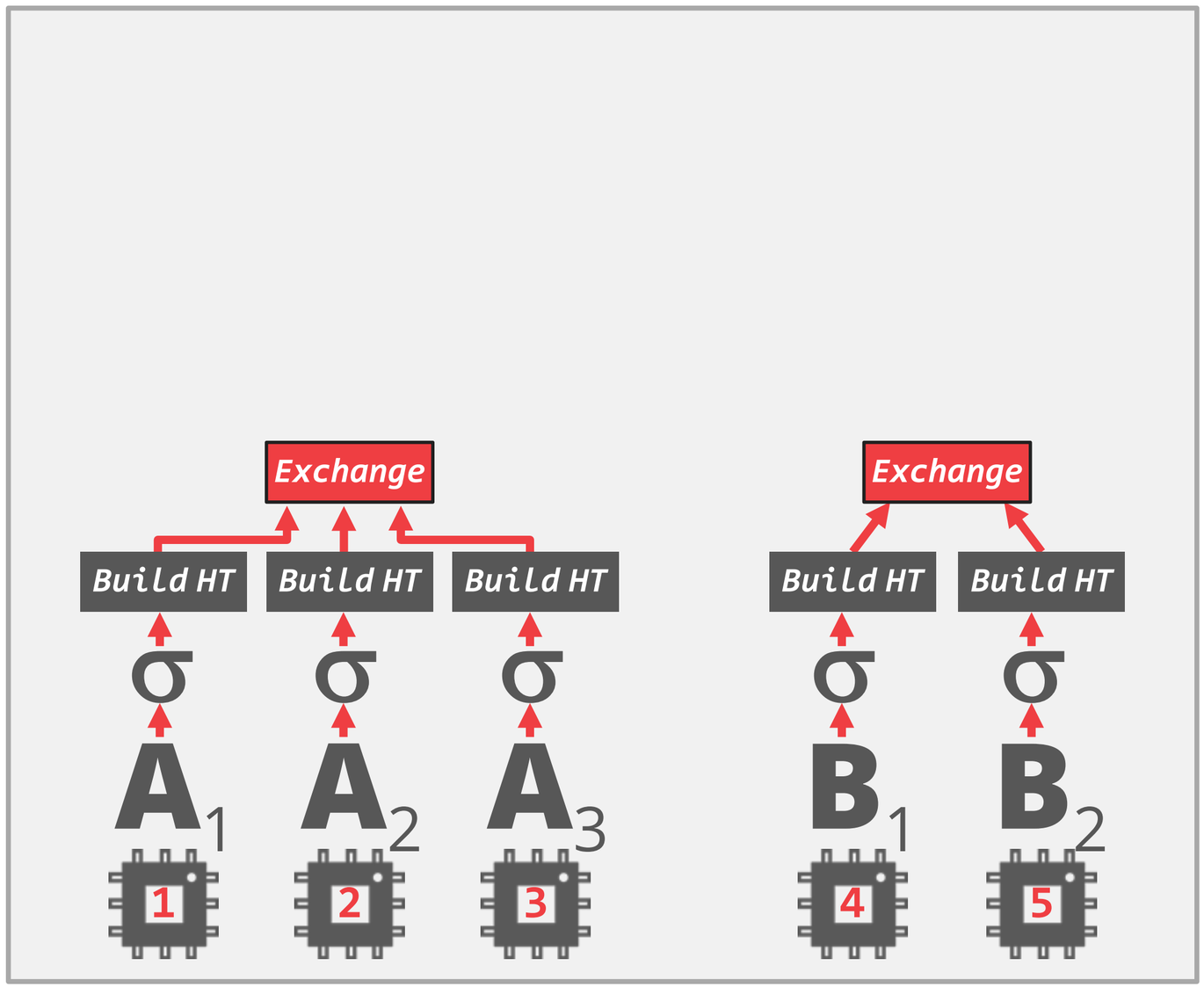




Intra-Operator Parallelism

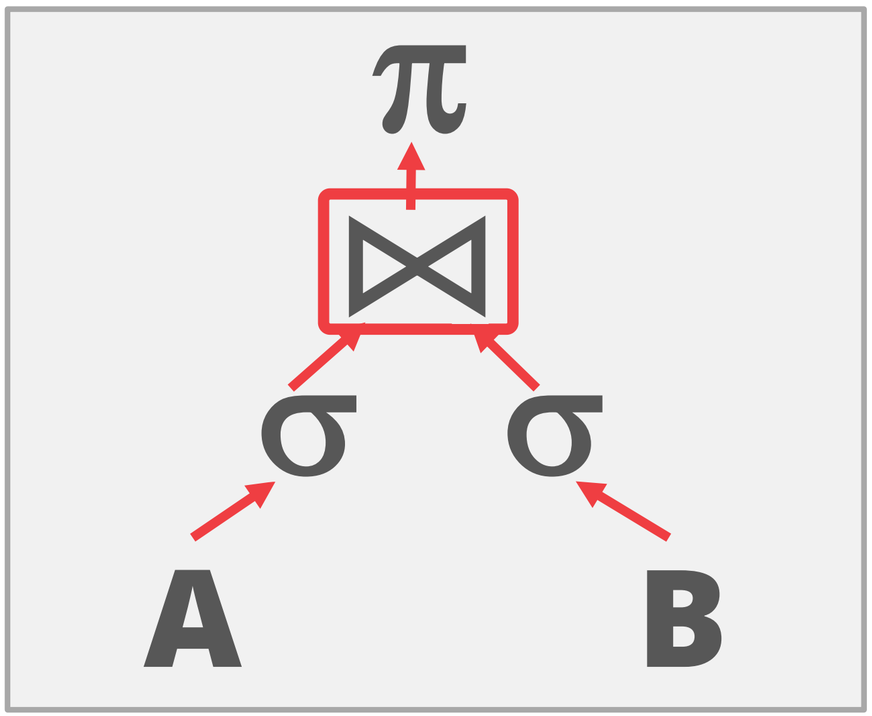

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

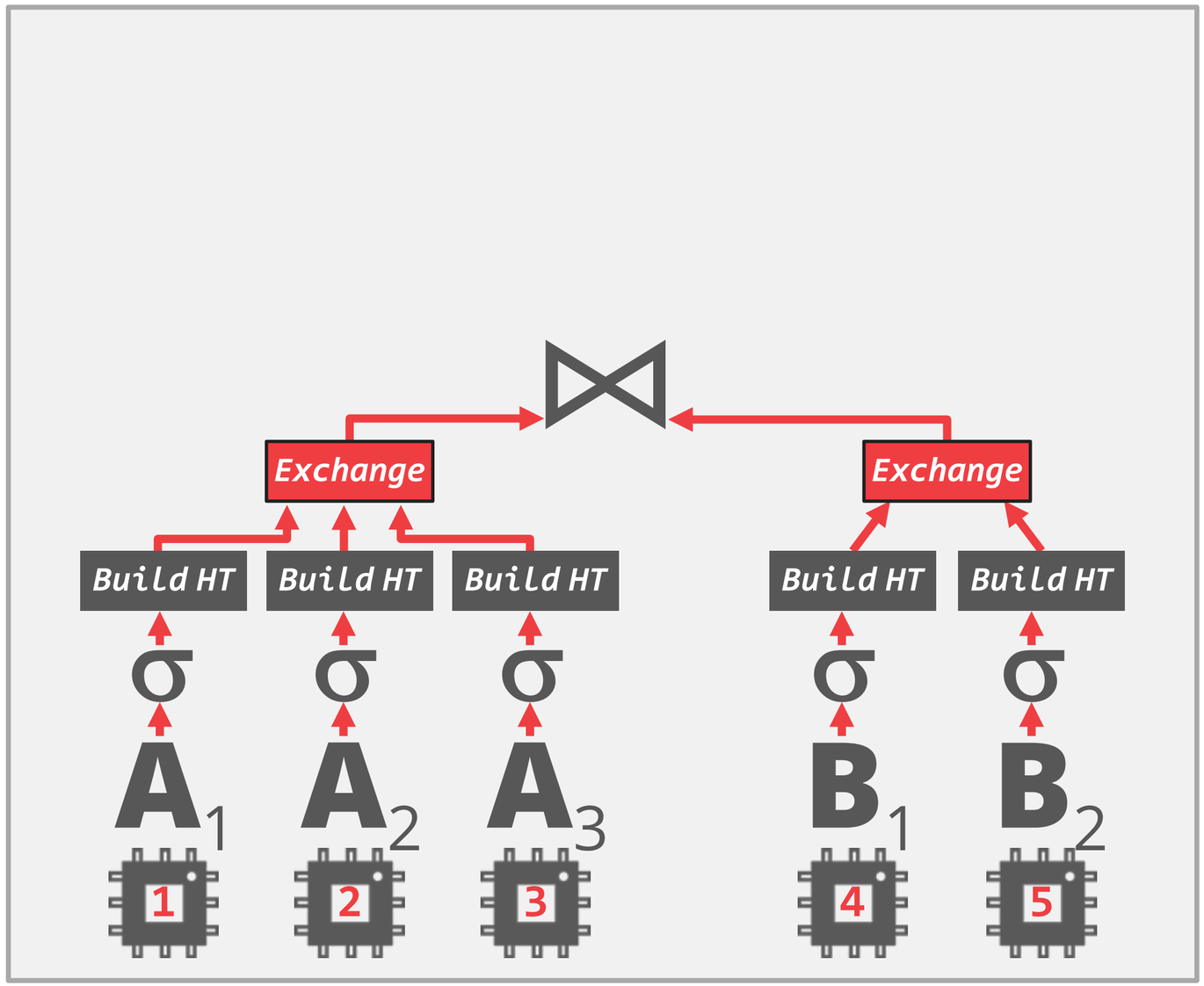




Intra-Operator Parallelism

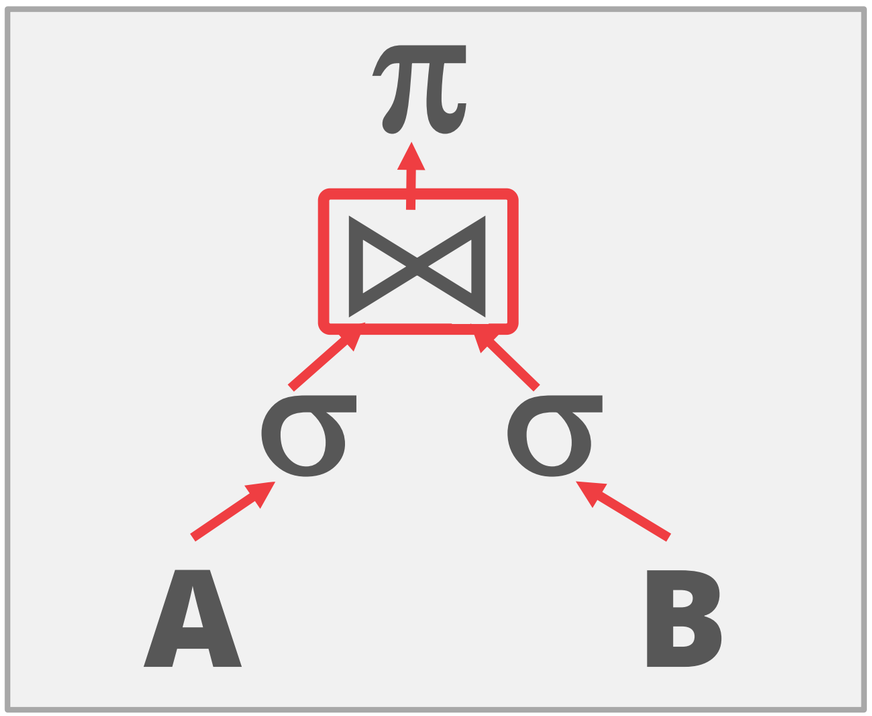
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

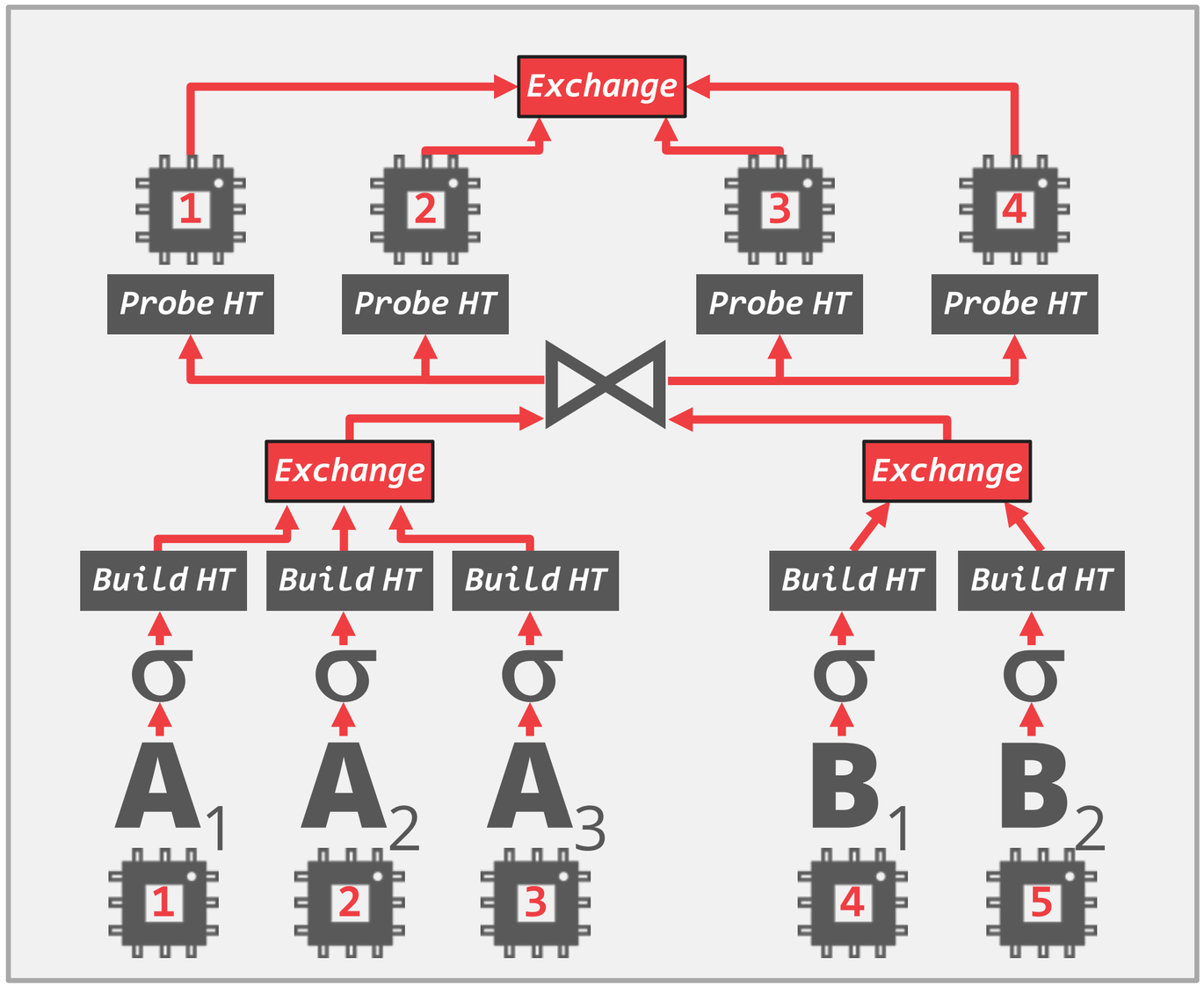




Intra-Operator Parallelism

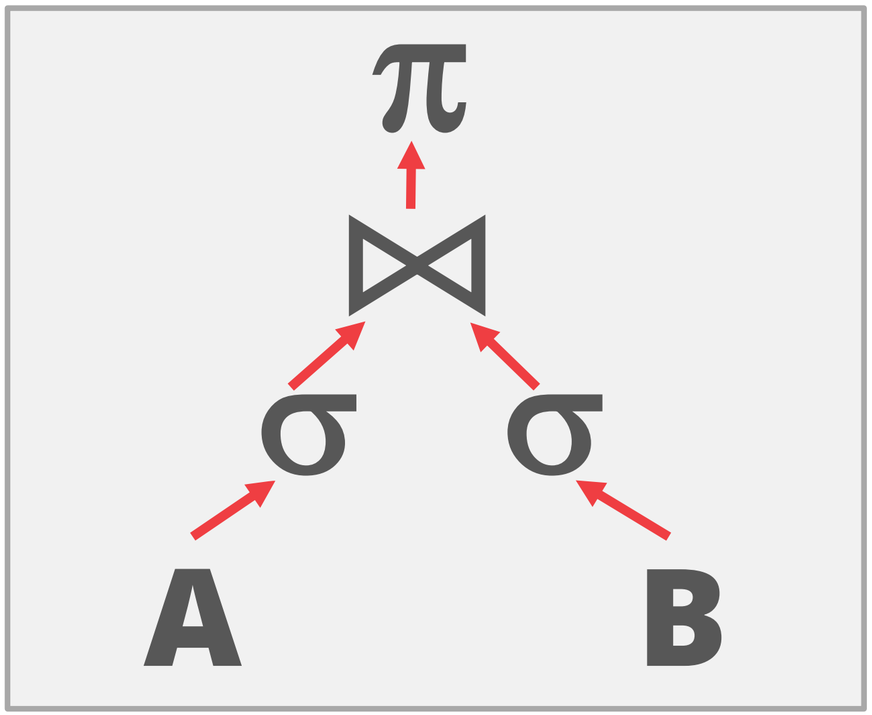

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```





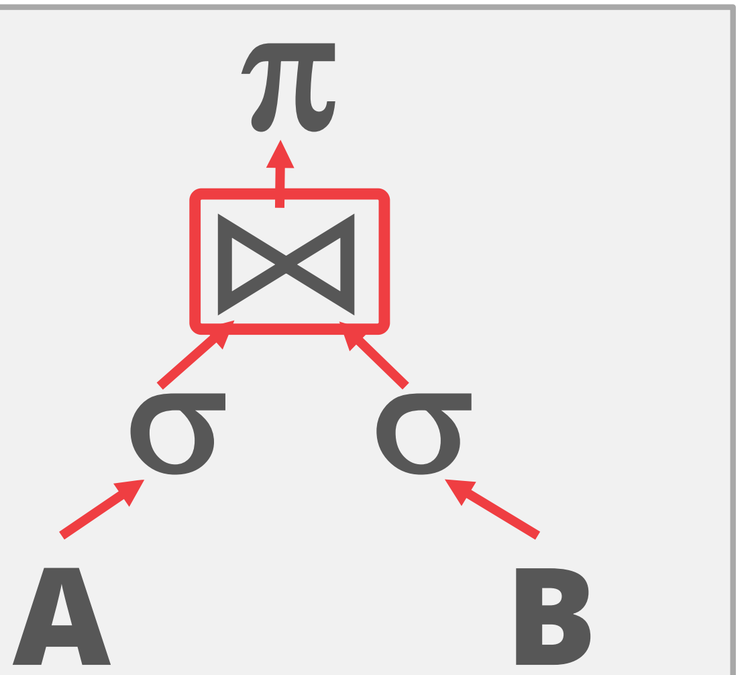
Inter-Operator Parallelism


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



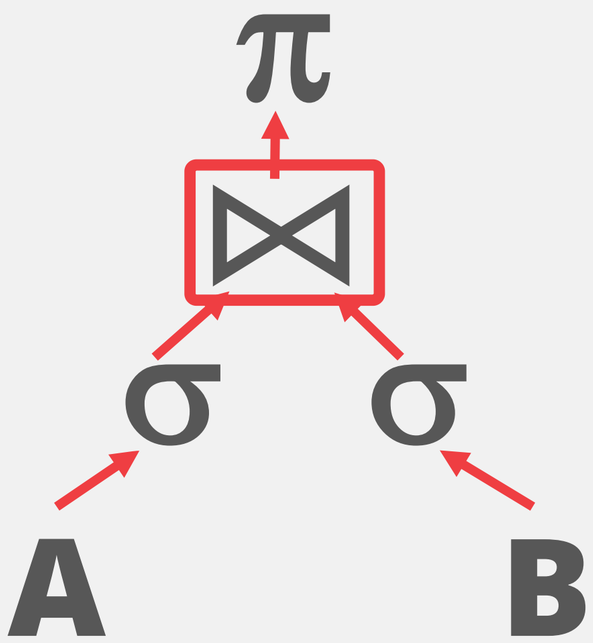
Inter-Operator Parallelism

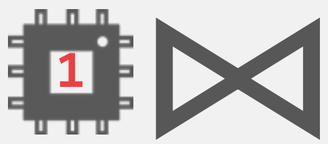

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



Inter-Operator Parallelism


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

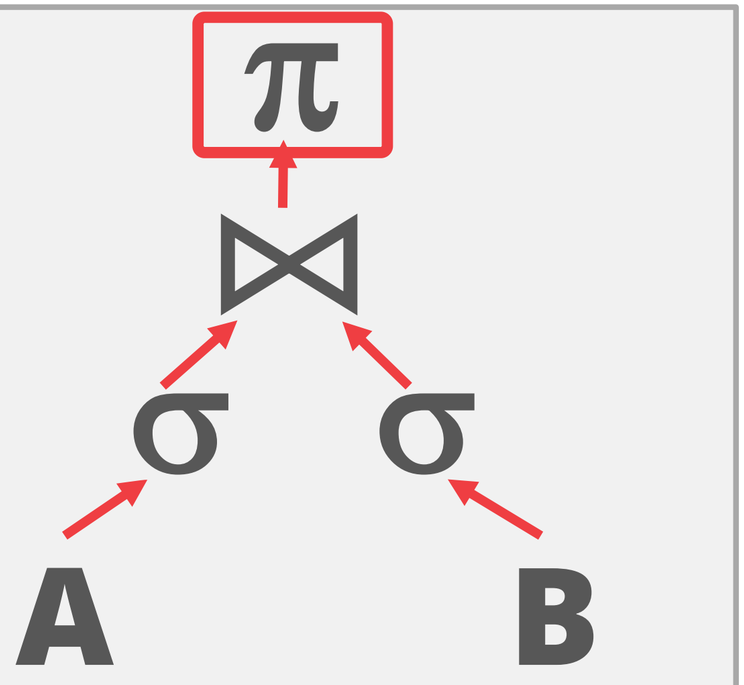


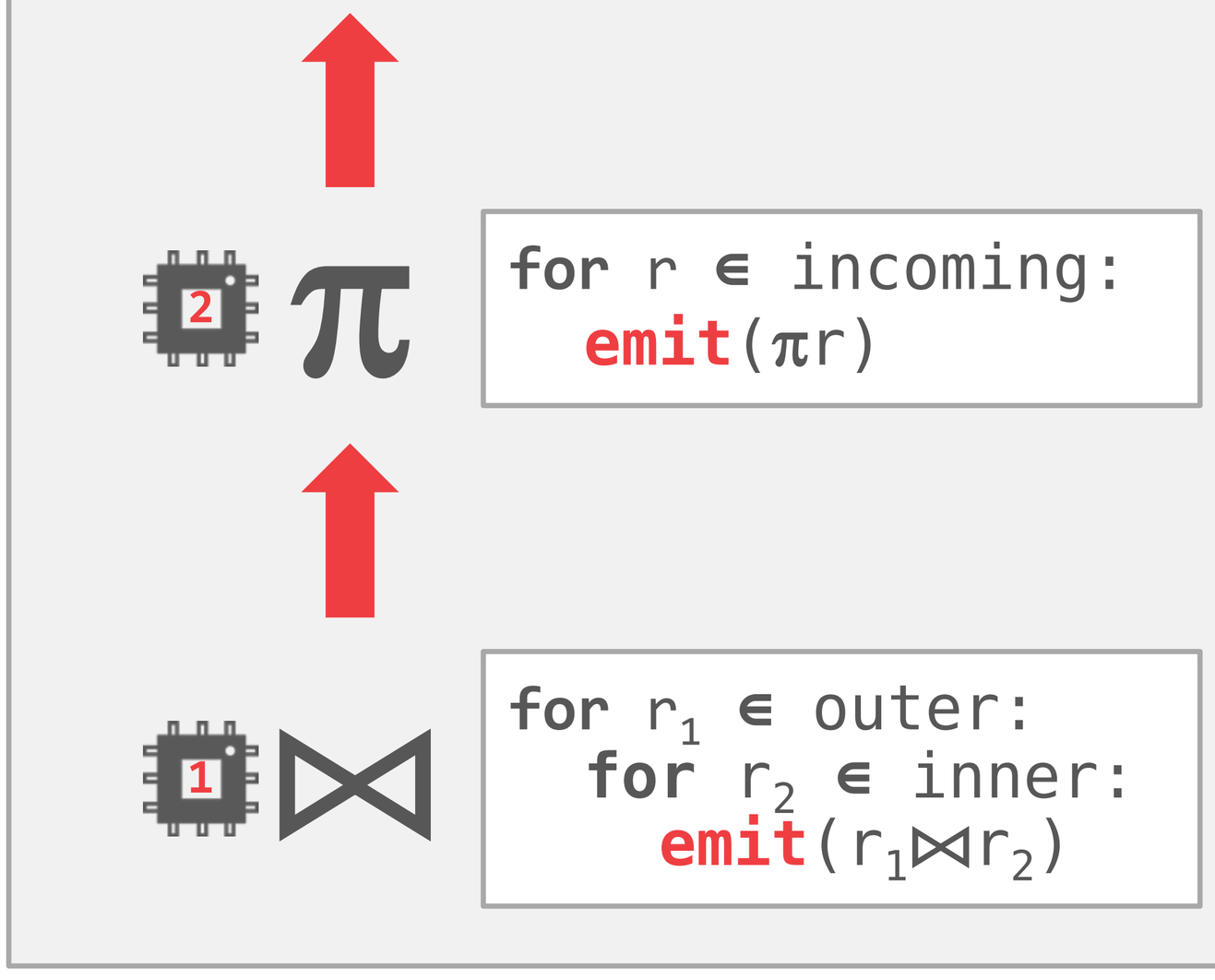



```
for r1 ∈ outer:
  for r2 ∈ inner:
    emit(r1⋈r2)
```

Inter-Operator Parallelism

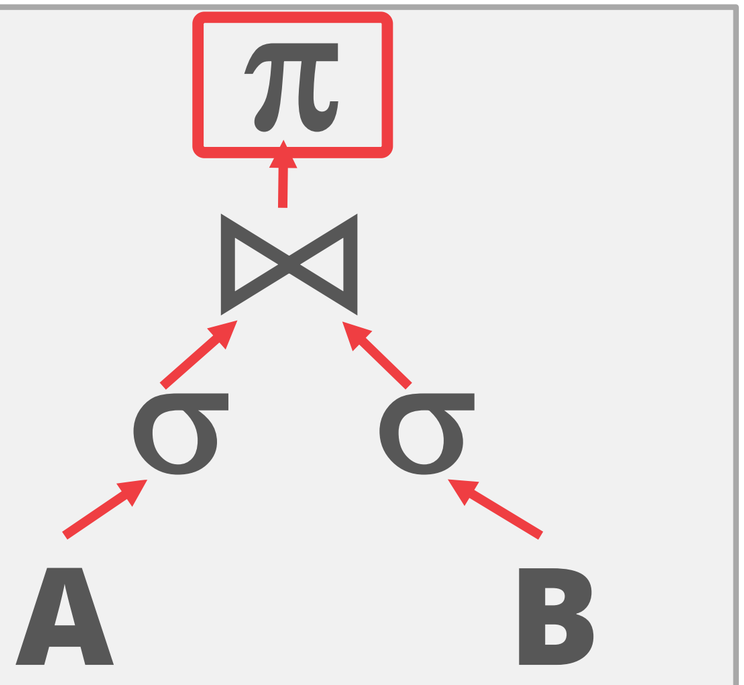

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

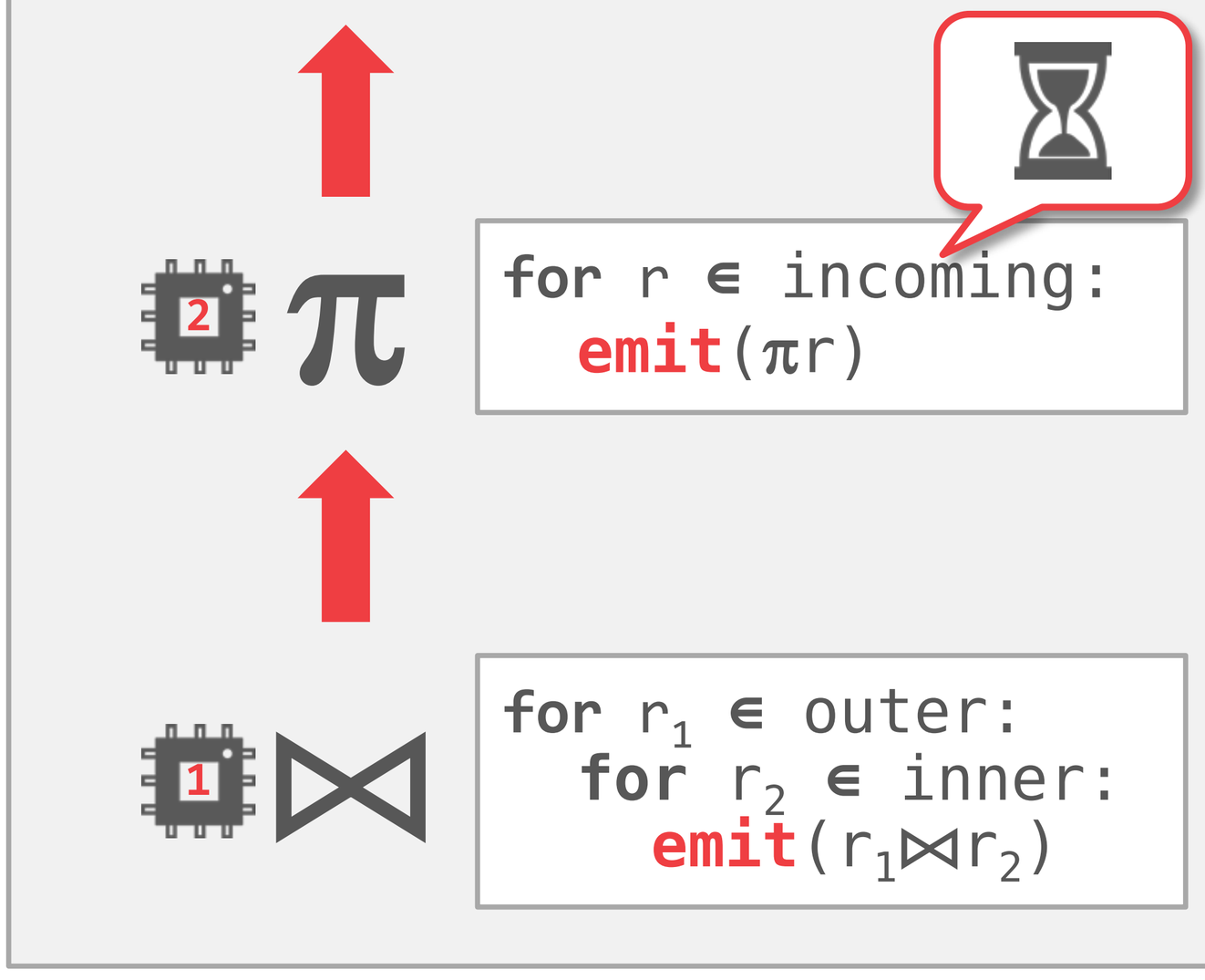




Inter-Operator Parallelism


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```





Morsel-Driven Scheduling [SIGMOD 2014]

- **Dynamic scheduling** of tasks that operate over ***horizontal partitions*** called "***morsels***" that are distributed across cores.
  - ***One worker*** per core
  - ***Pull-based*** task assignment
  - ***Round-robin*** data placement

MORSEL-DRIVEN PARALLELISM: A NUMA-AWARE QUERY EVALUATION FRAMEWORK FOR THE MANY-CORE AGE
SIGMOD 2014

- **Supports parallel, *NUMA-aware* operator implementations.**

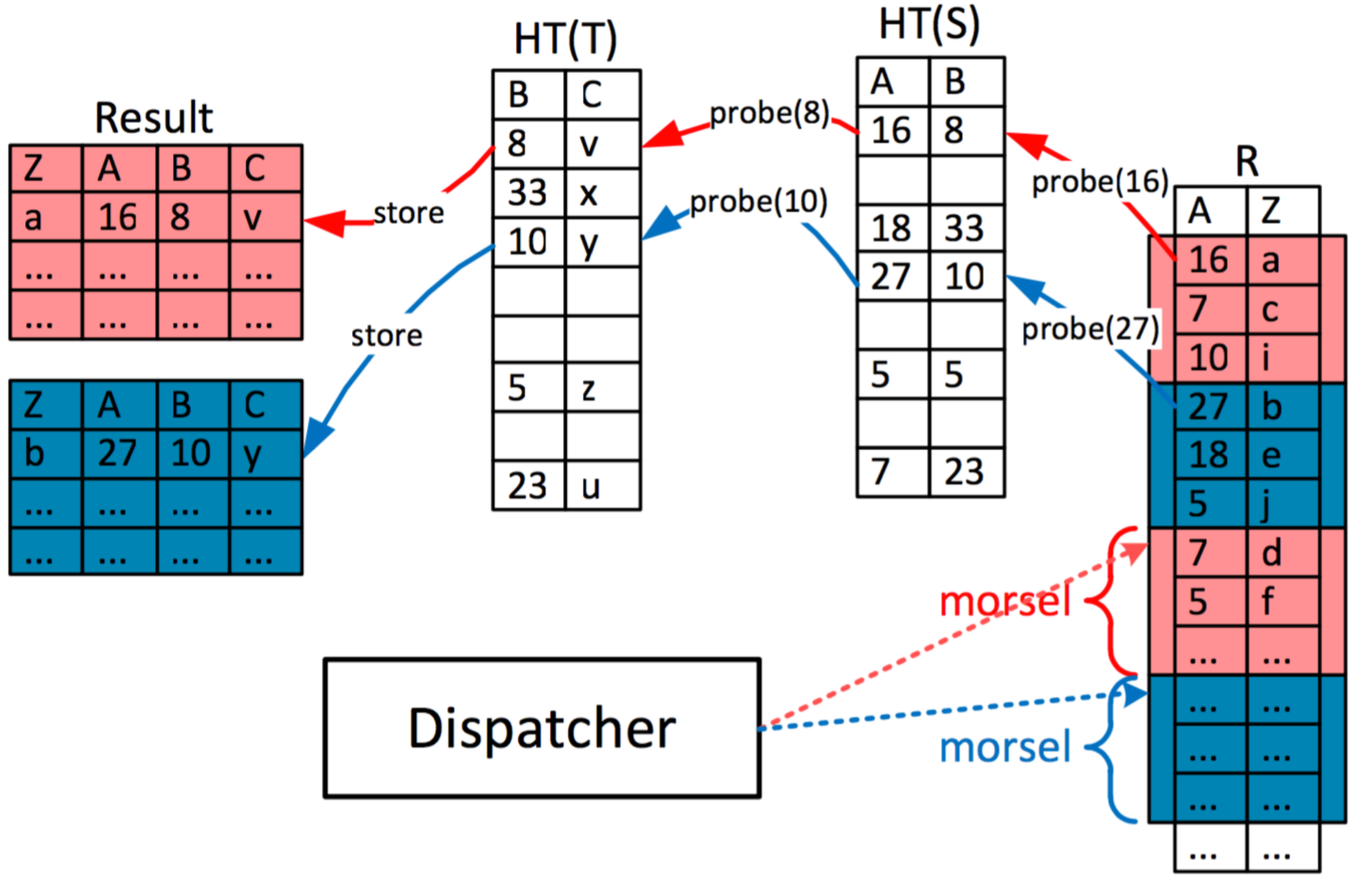
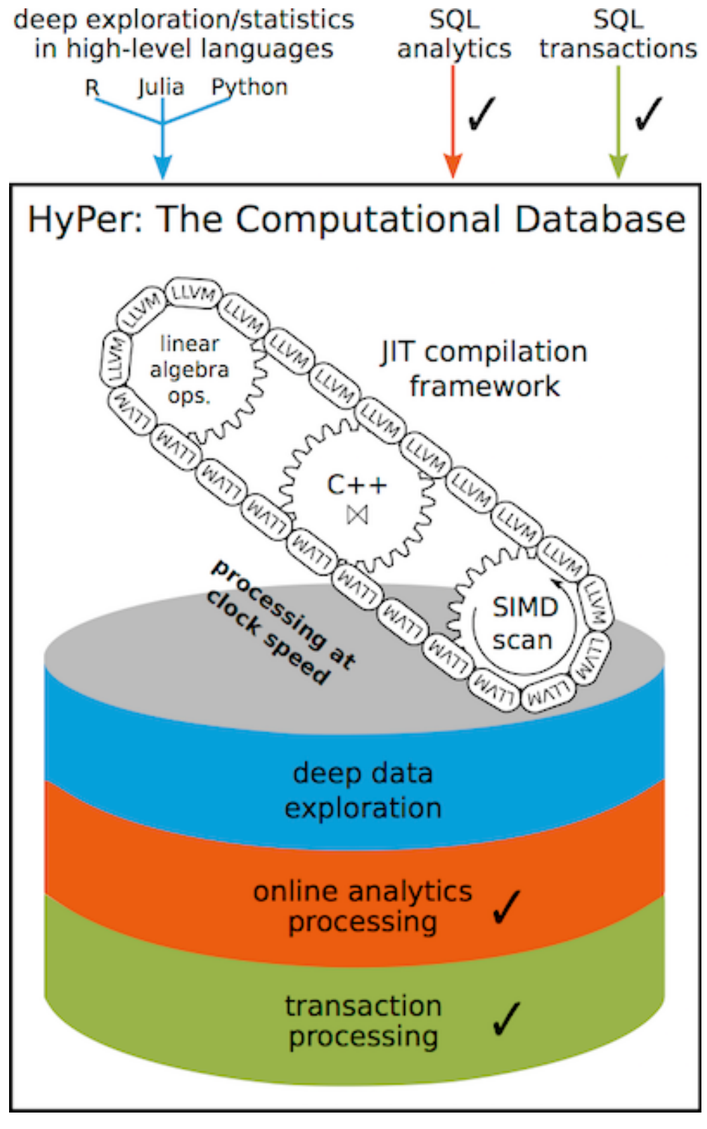


Figure 1: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$

HyPer: Architecture

- No separate (centralized) *dispatcher* thread.
- The threads perform *cooperative scheduling* for each query plan.
  - Each worker has a queue of tasks that will execute on *morsels* that are local to it.
  - It pulls the next task from a global work queue.



Morsel-Driven Execution

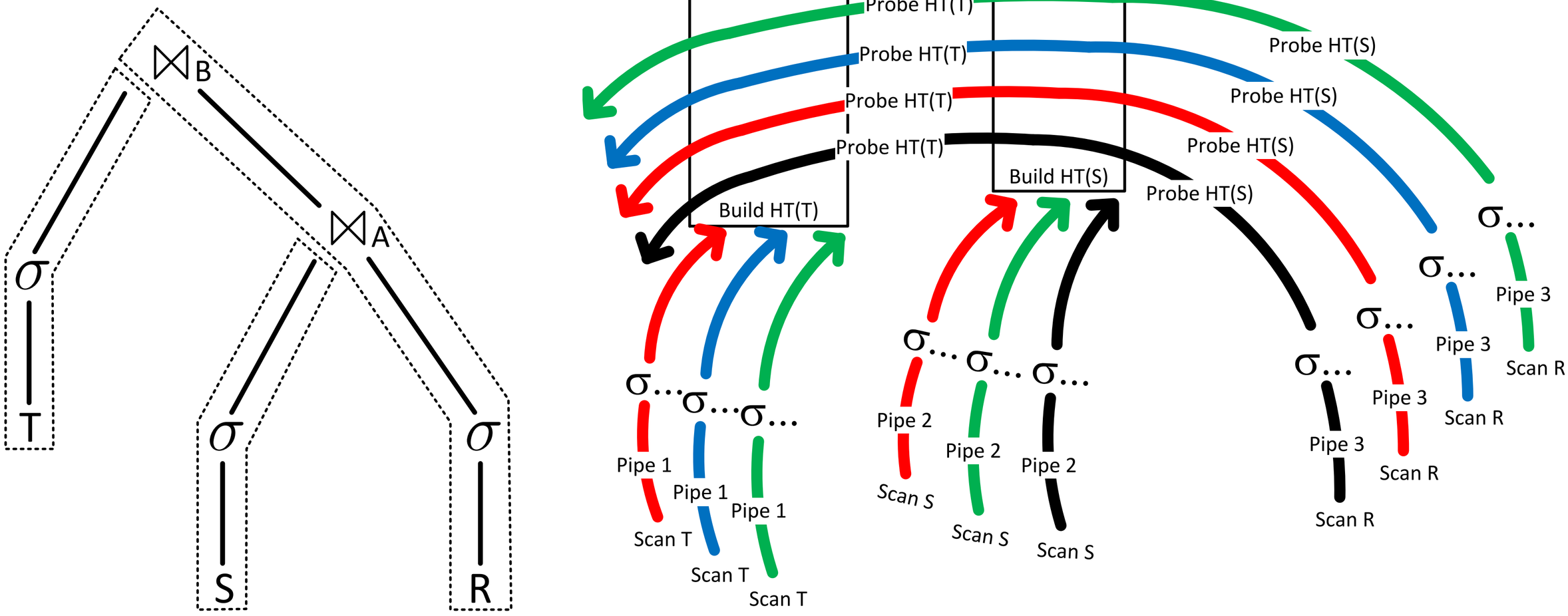
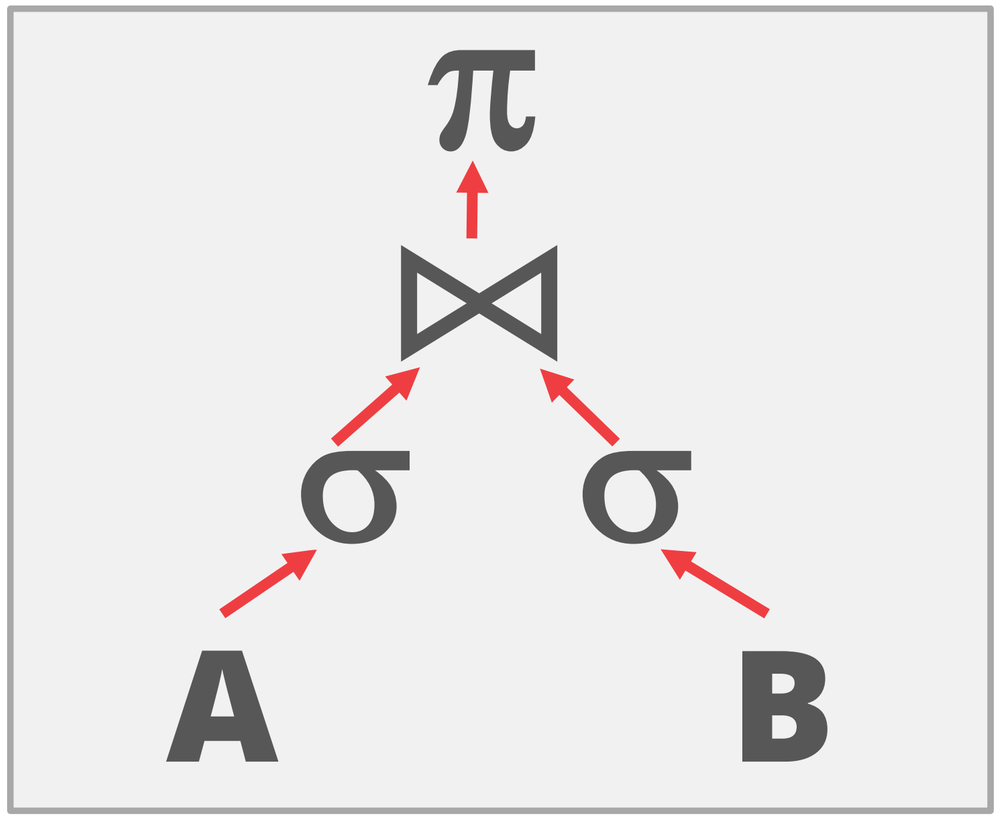


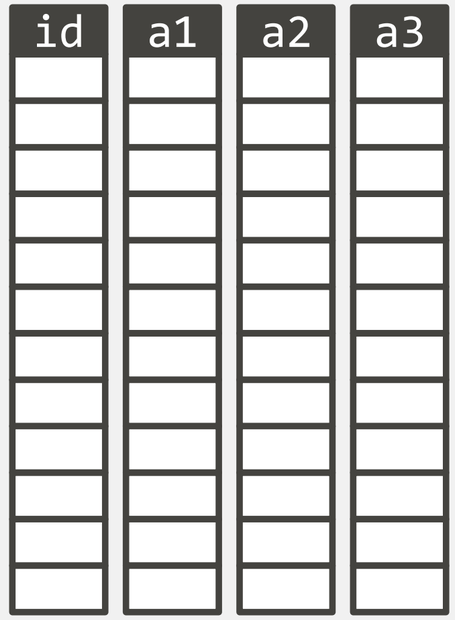
Figure 2: Parallellizing the three pipelines of the sample query plan: (left) algebraic evaluation plan; (right) three- respectively four-way parallel processing of each pipeline

HyPer: Data Partitioning


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

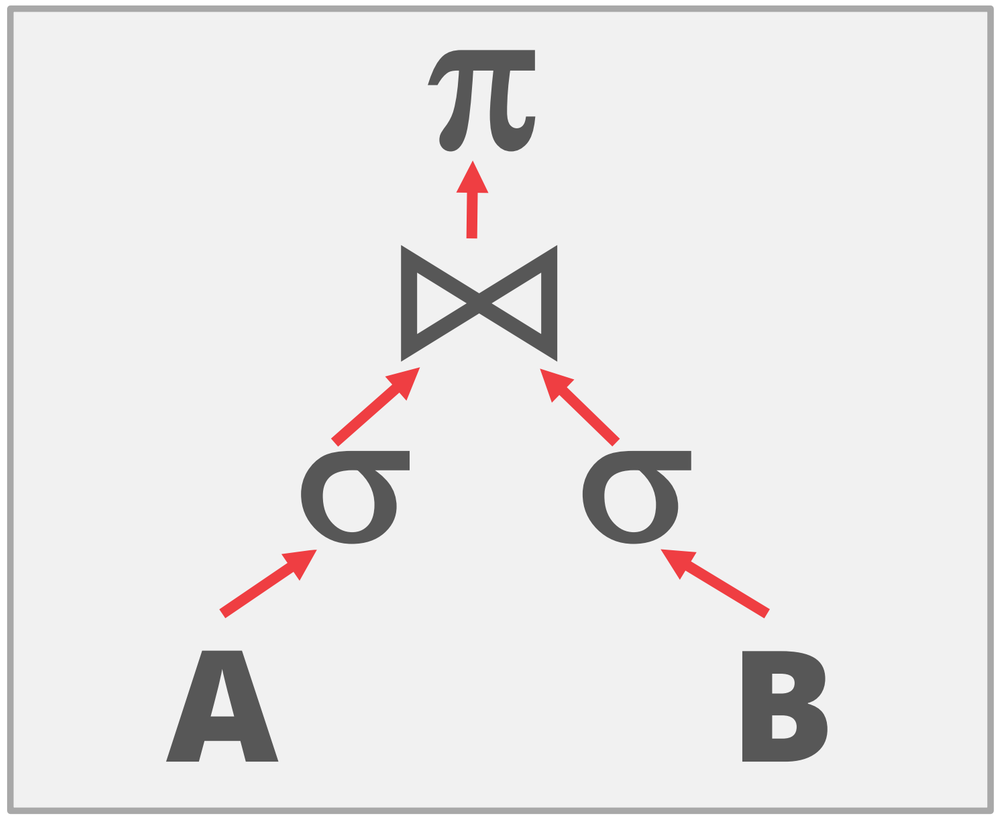


Data Table



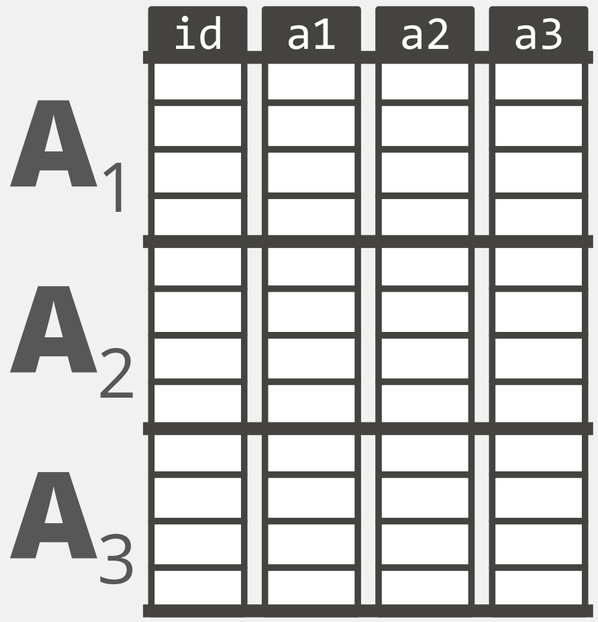
HyPer: Data Partitioning


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



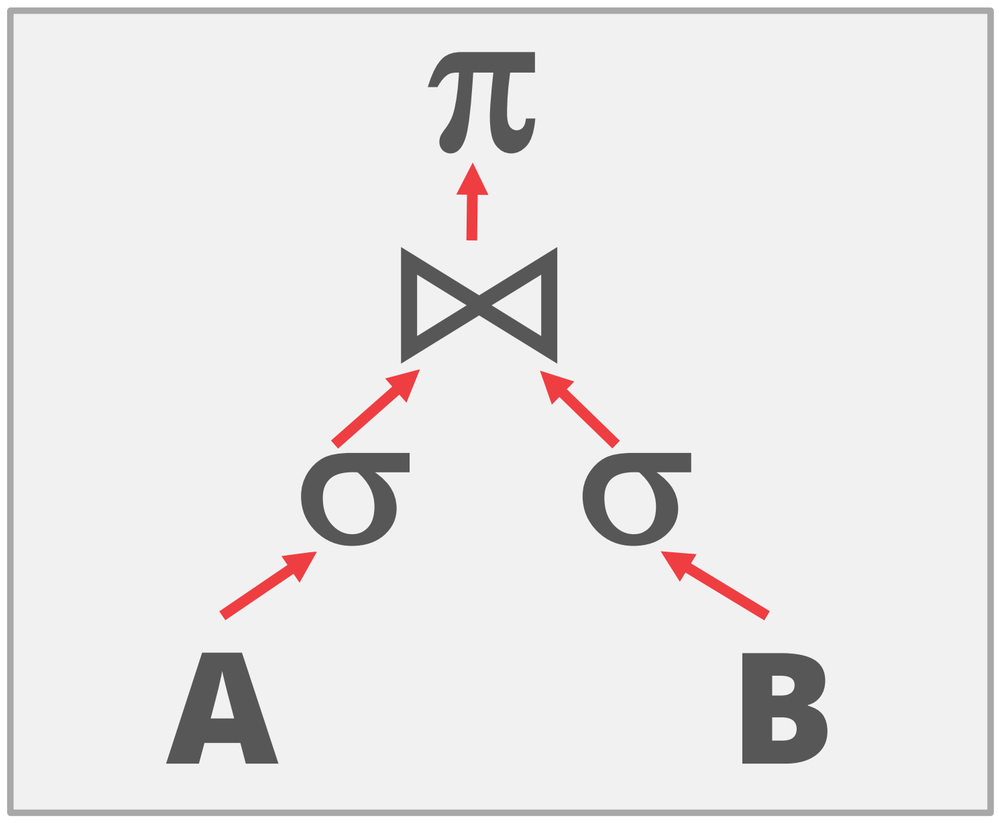
Data Table

Morsels



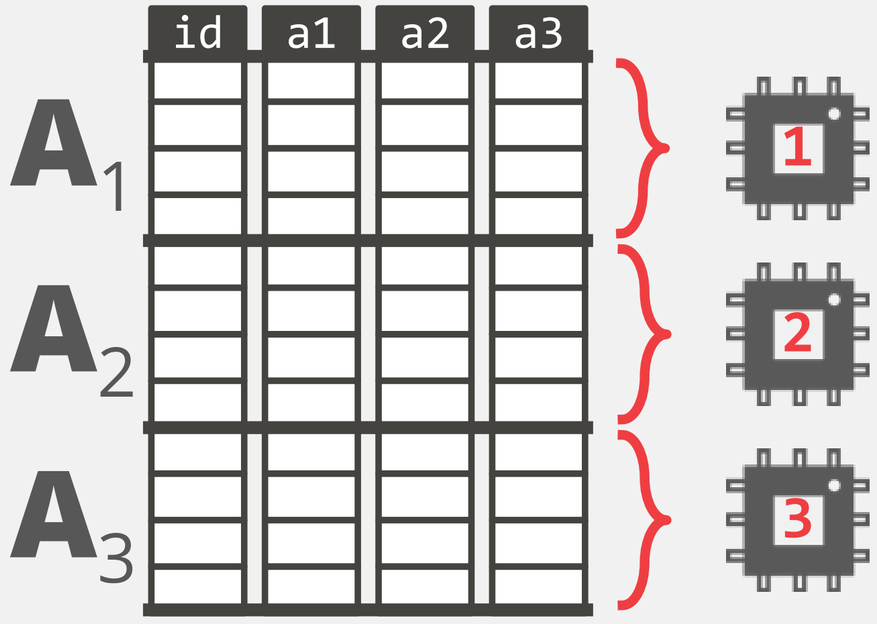
HyPer: Data Partitioning


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



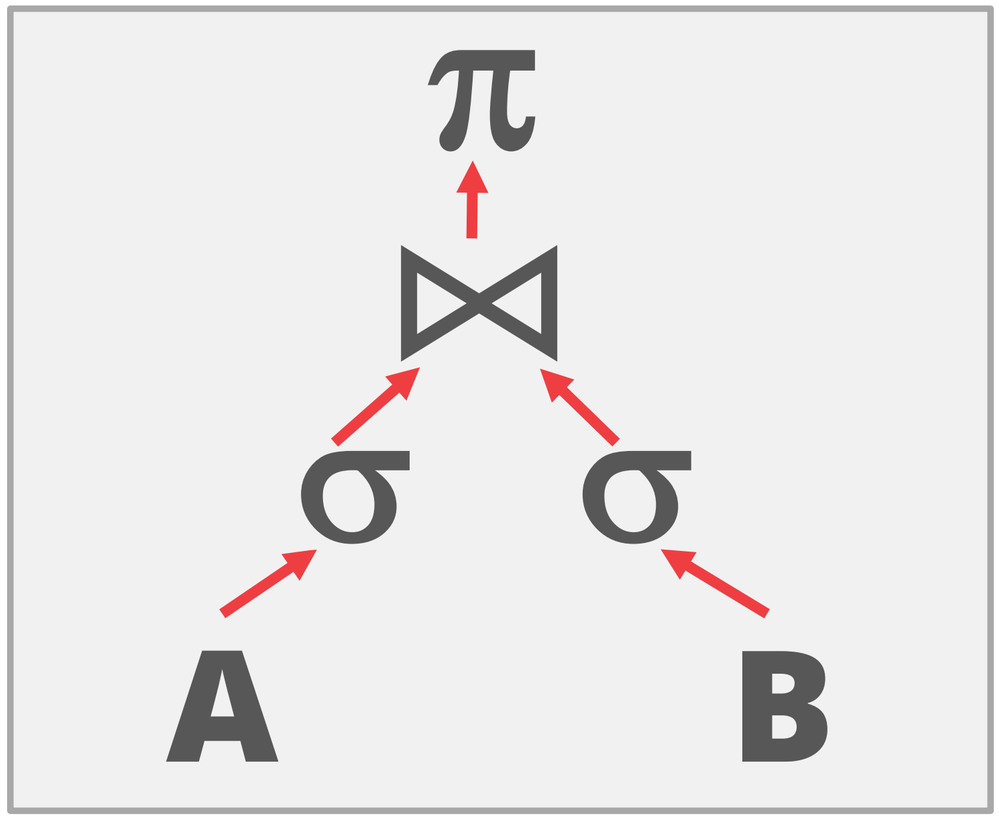
Data Table

Morsels

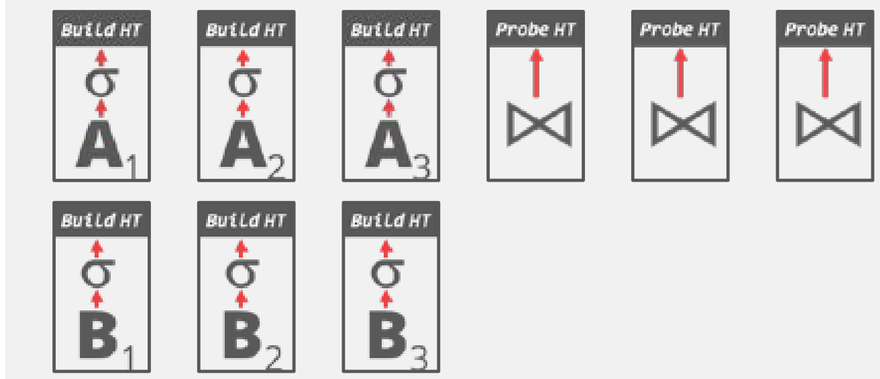


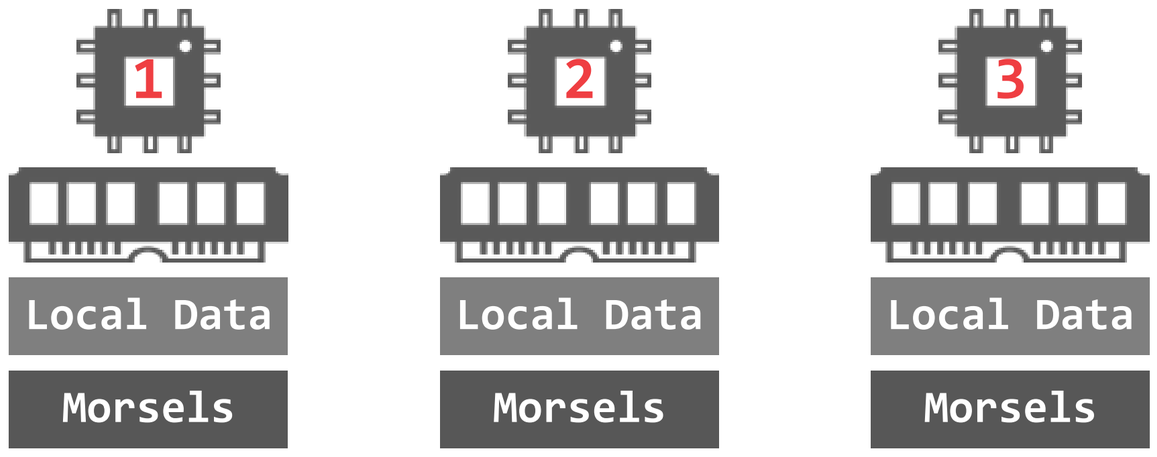
HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



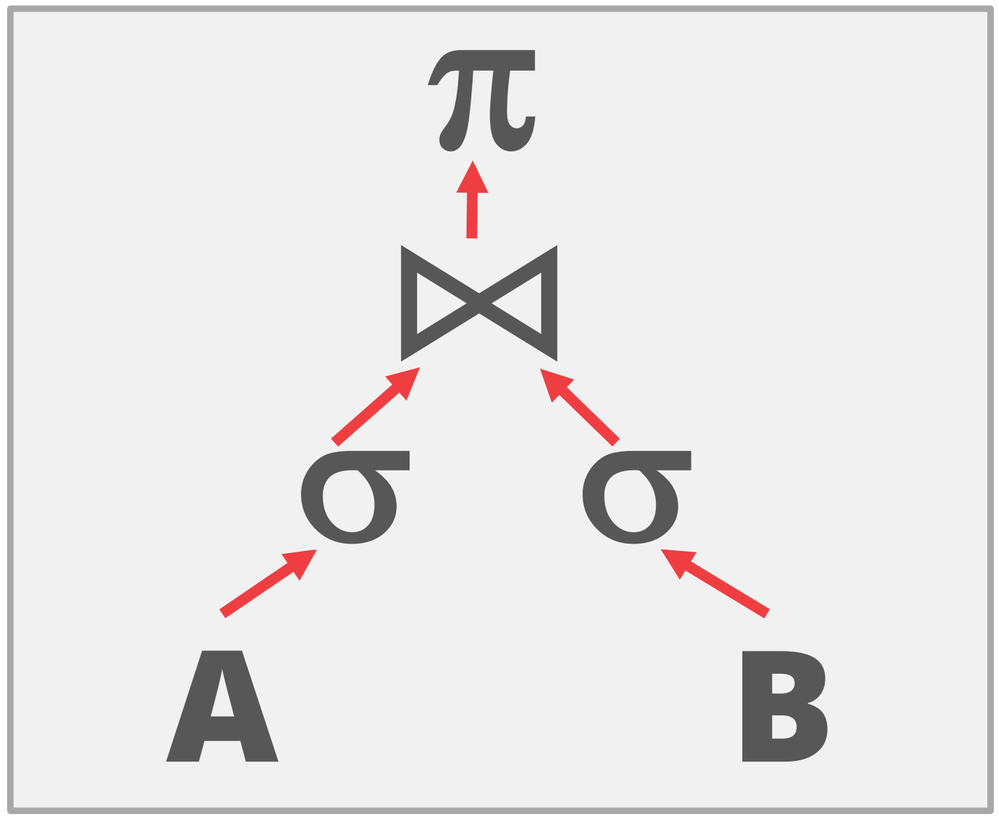
Task Queues



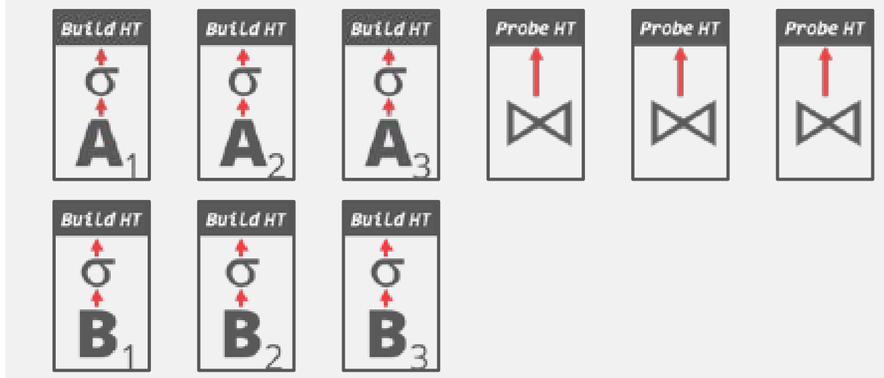


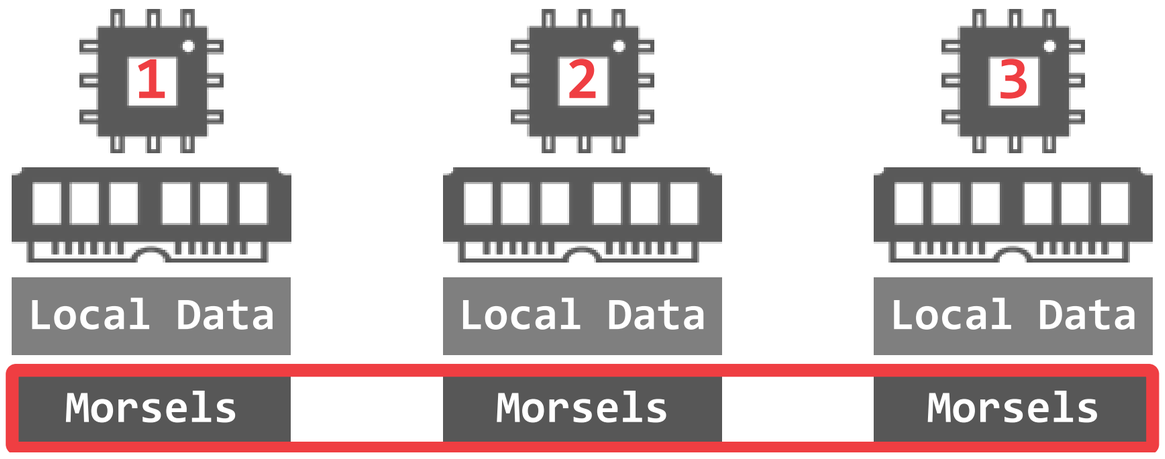
HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



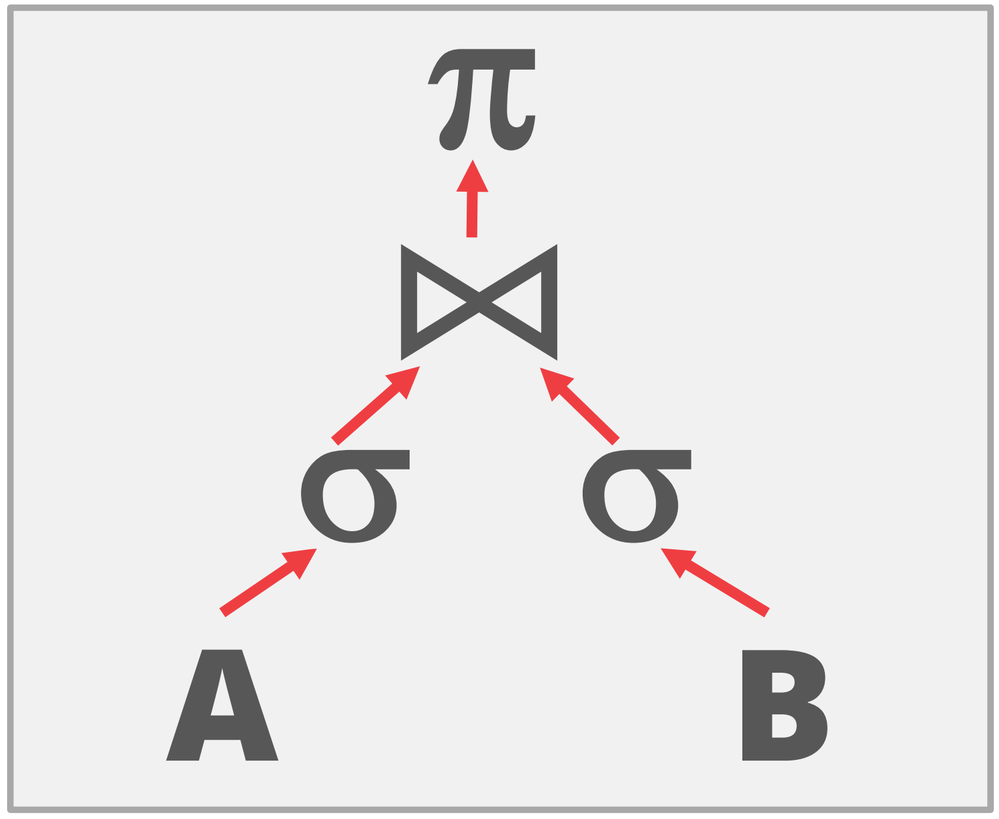
Task Queues



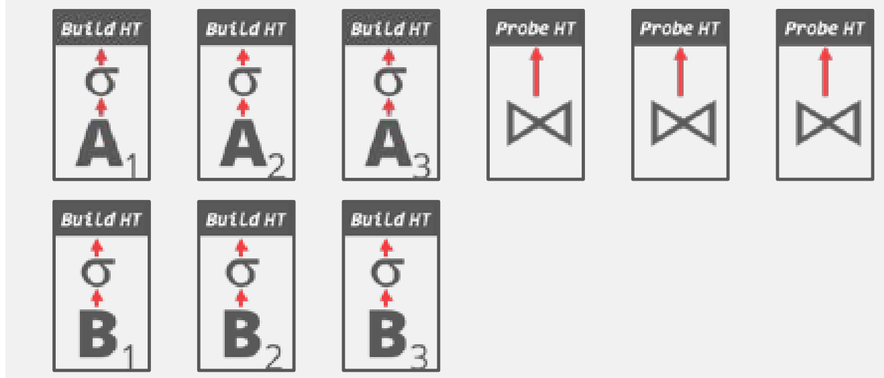


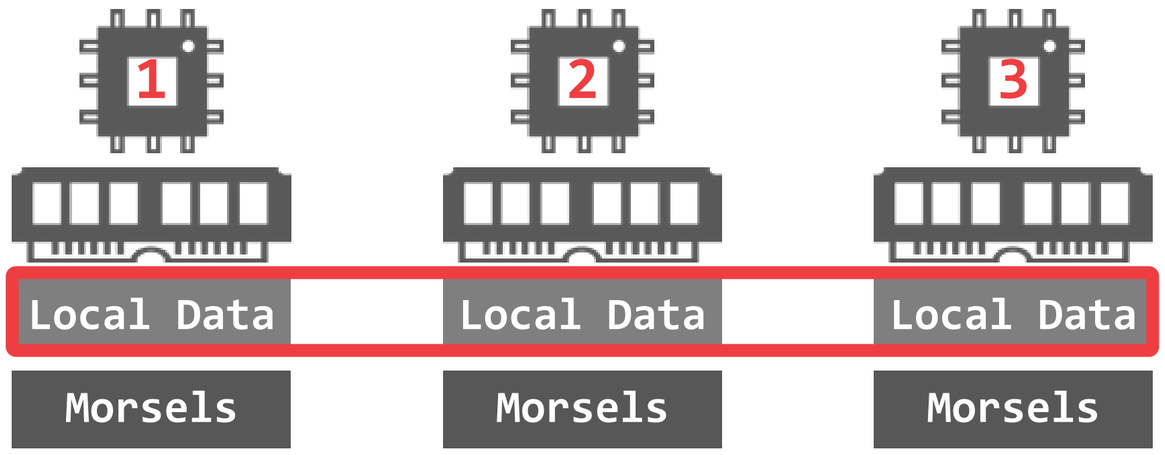
HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



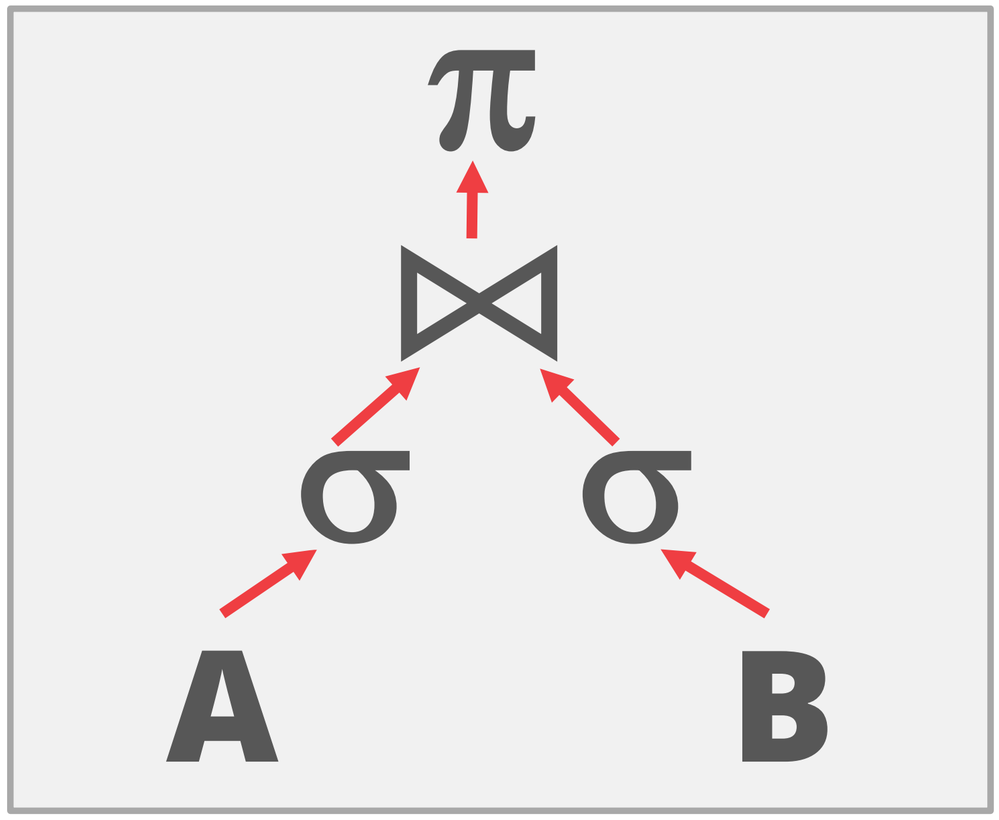
Task Queues



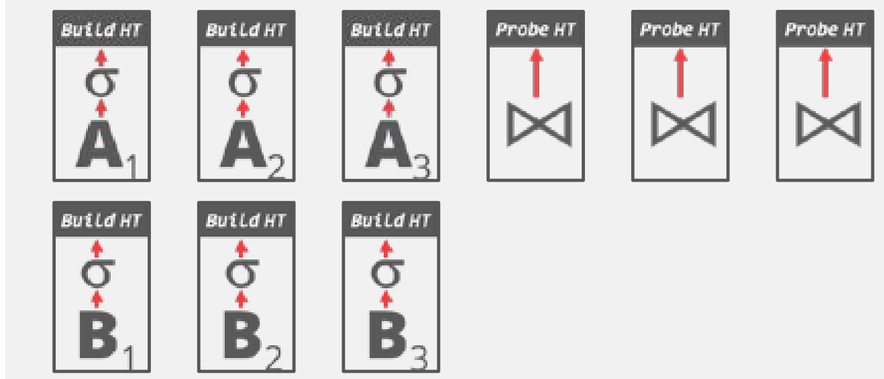


HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

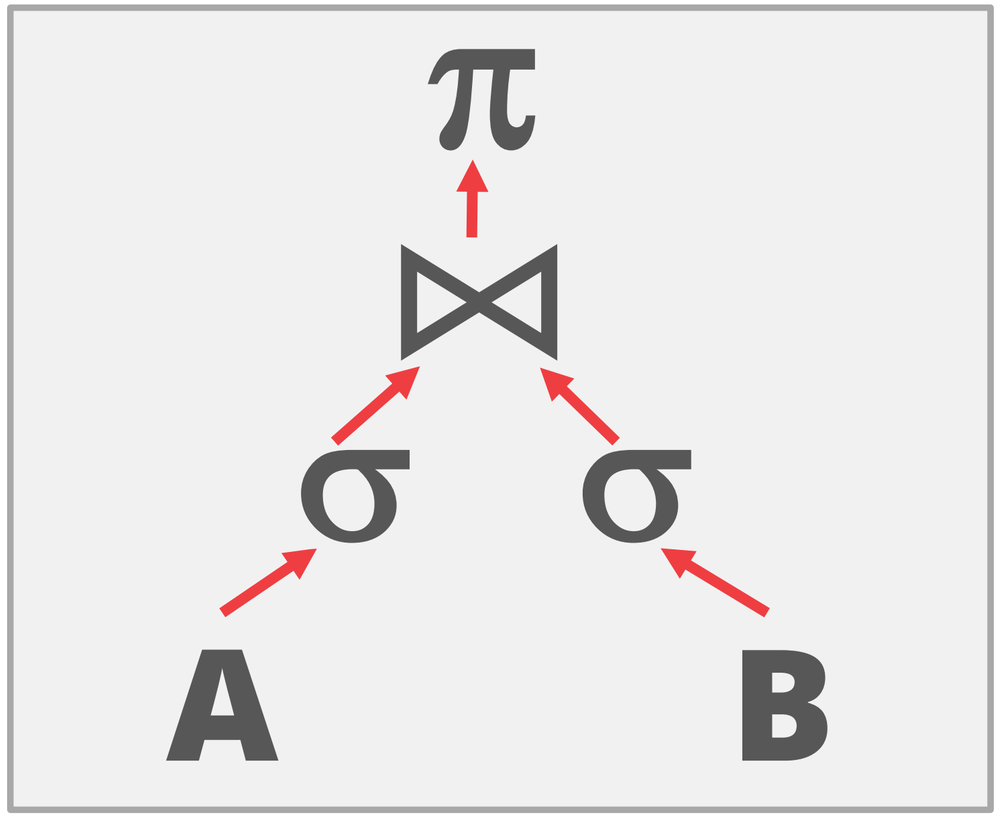


Task Queues

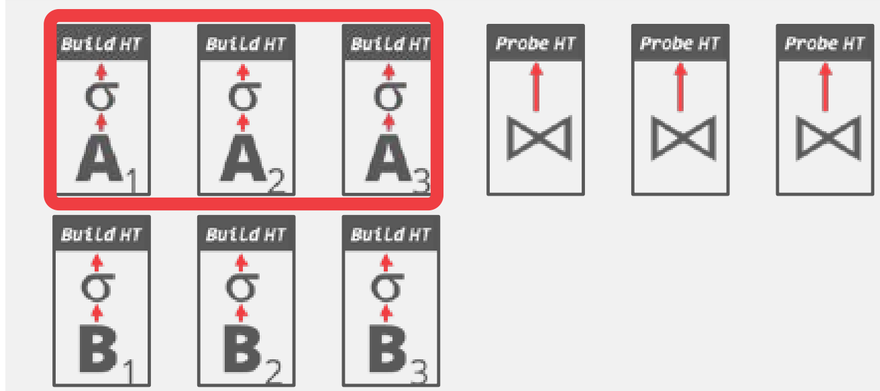


HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

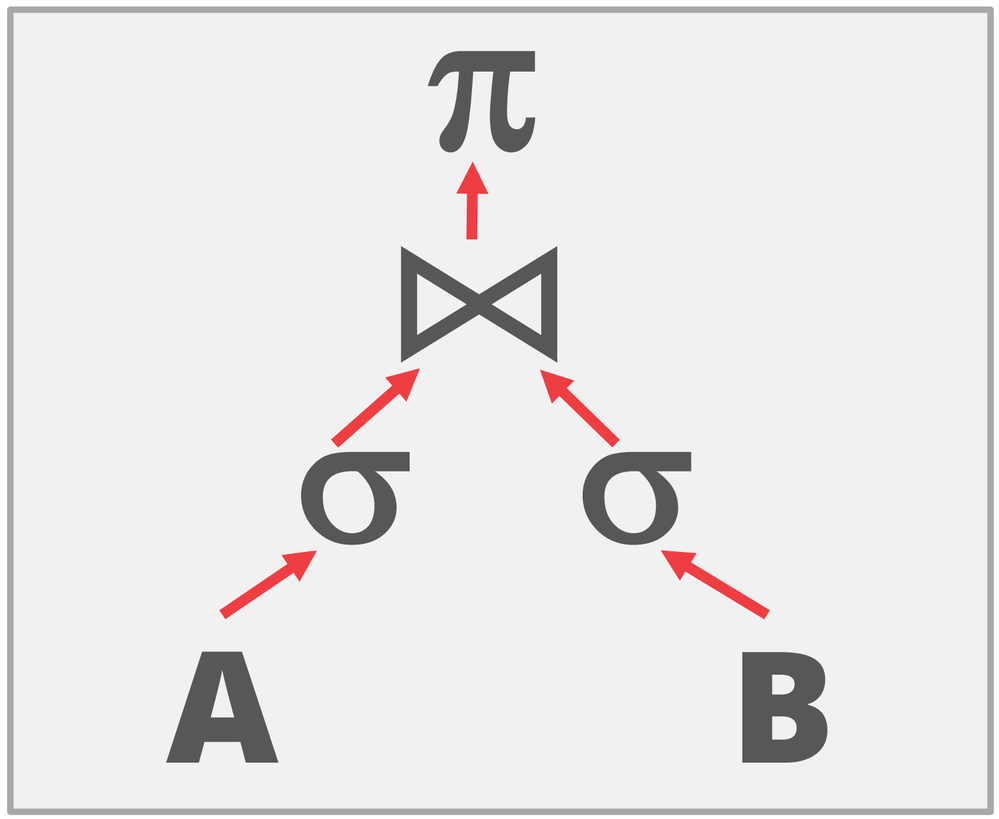


Task Queues

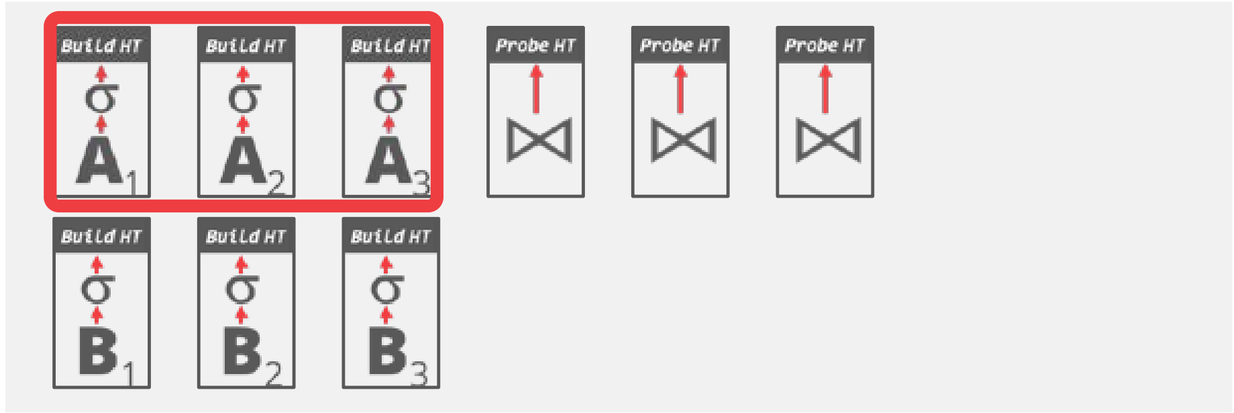


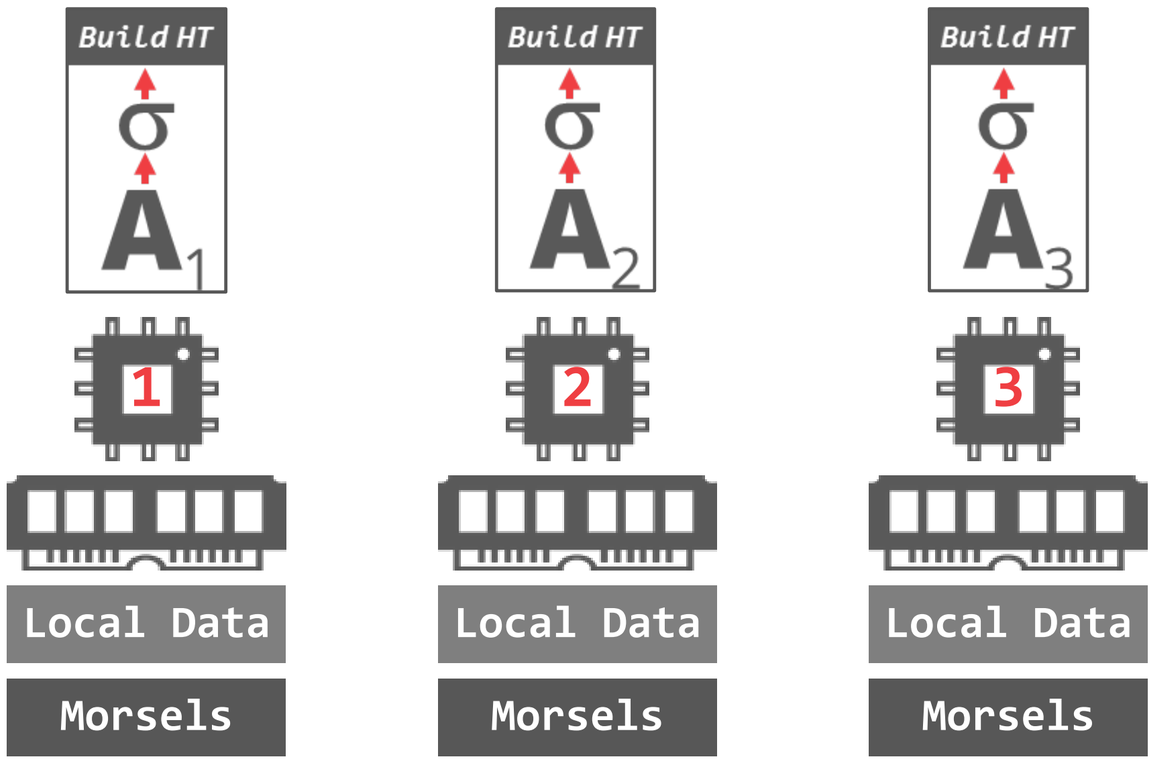
HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



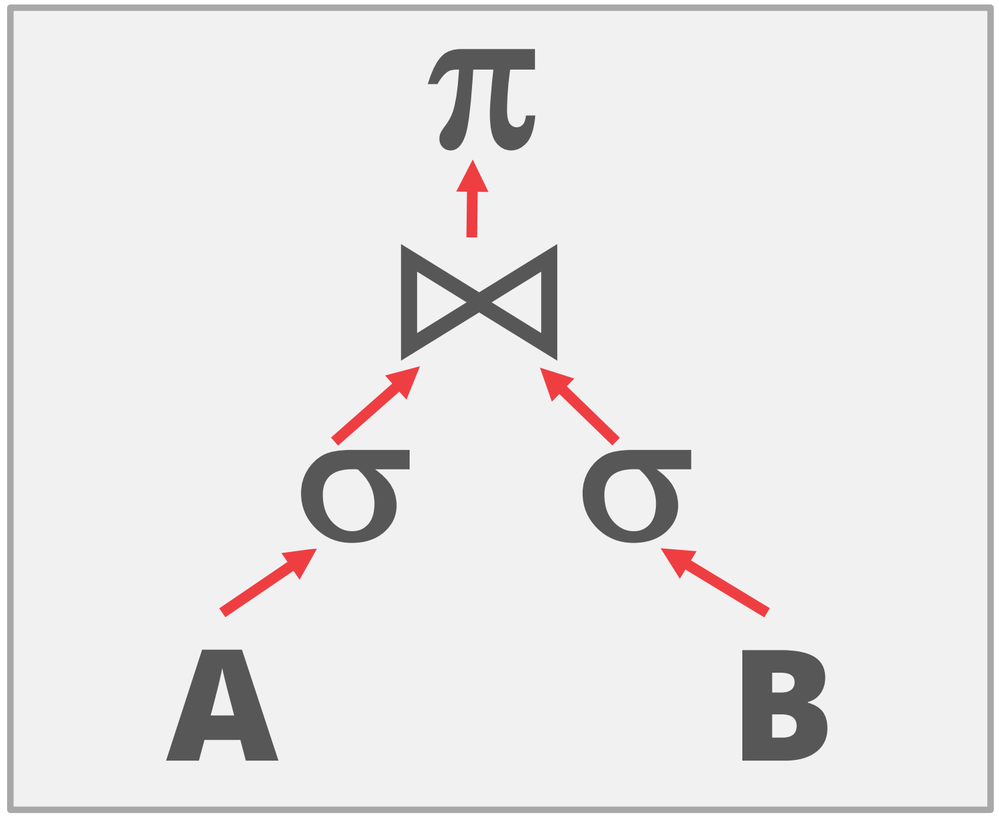
Task Queues



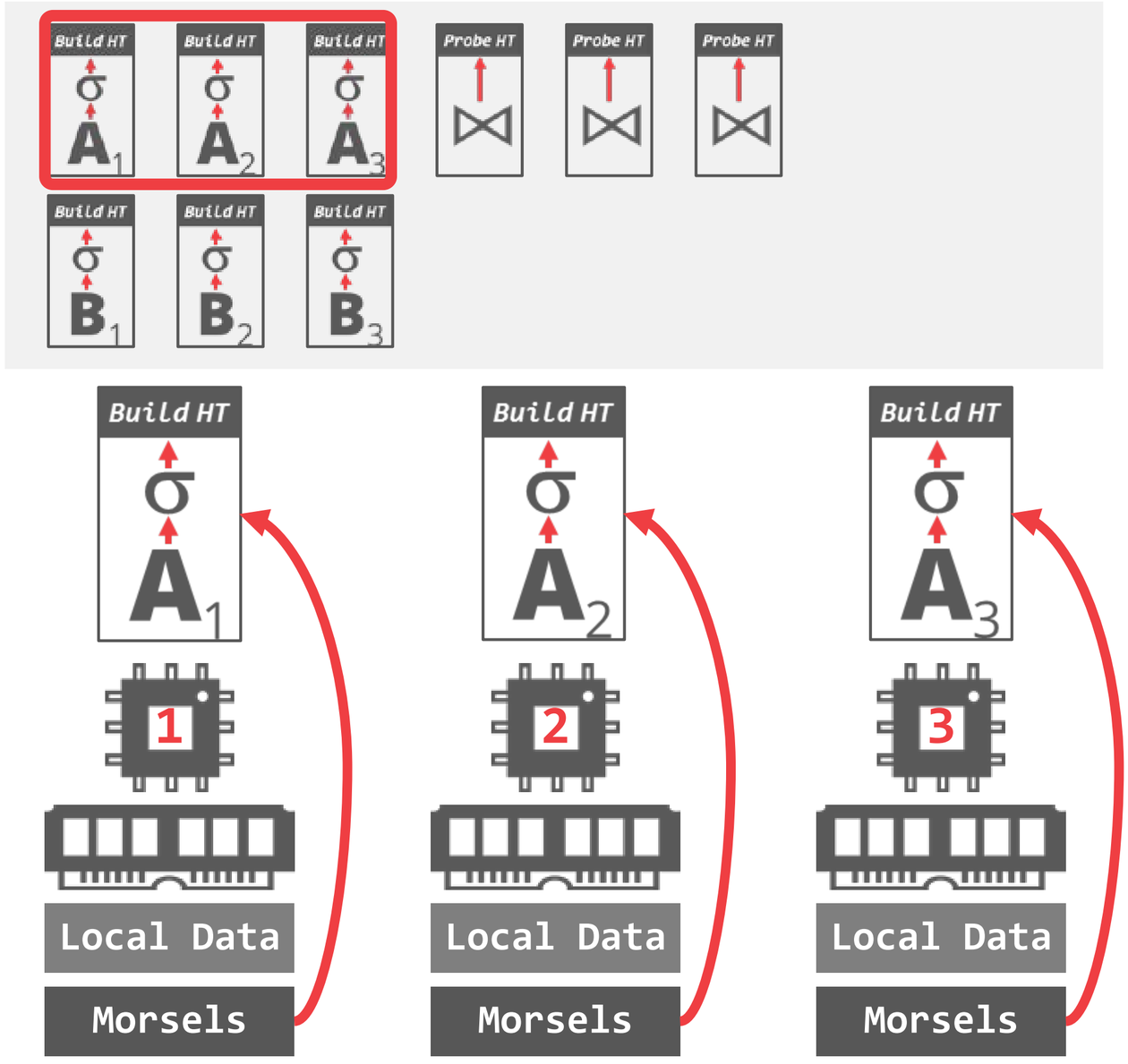


HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

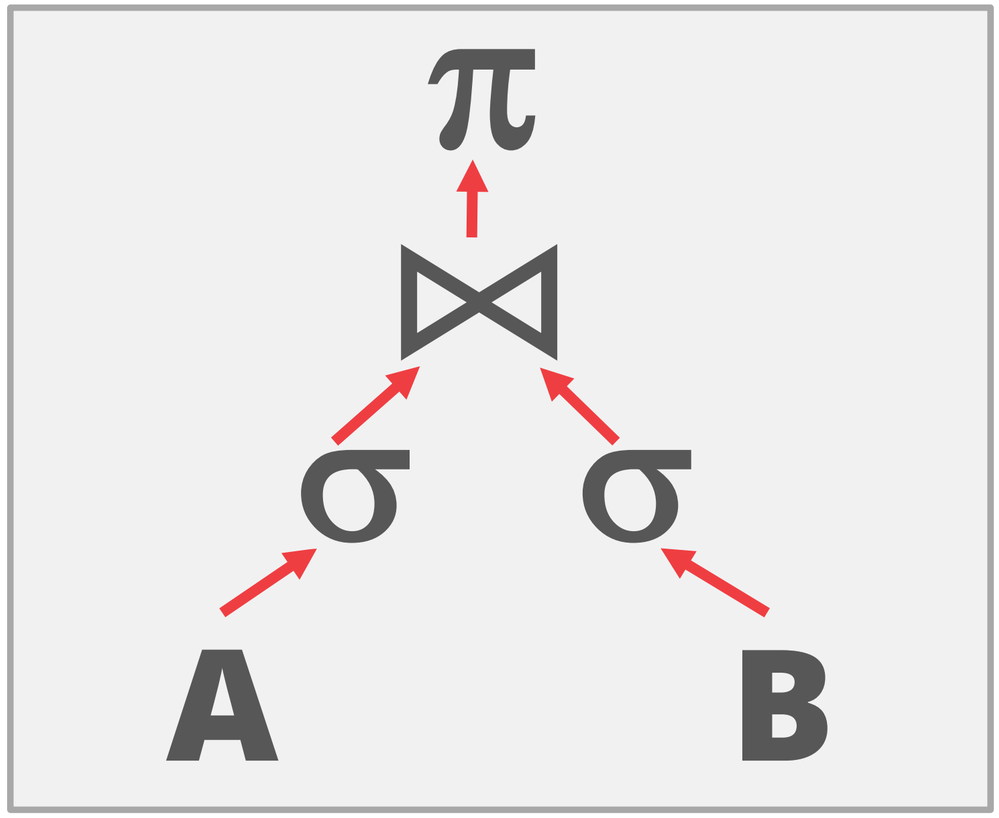


Task Queues

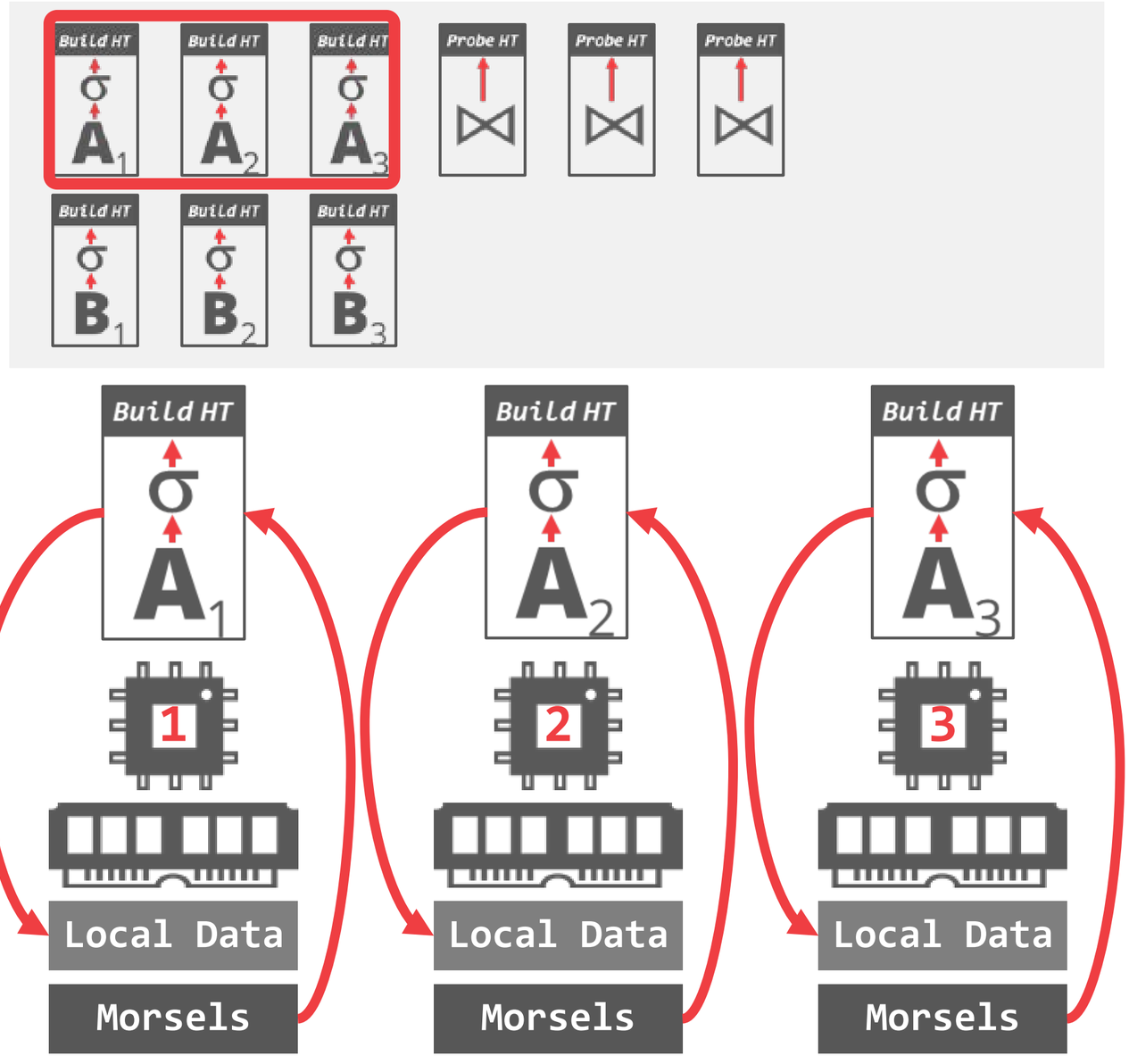


HyPer: Morsel-Driven Execution

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

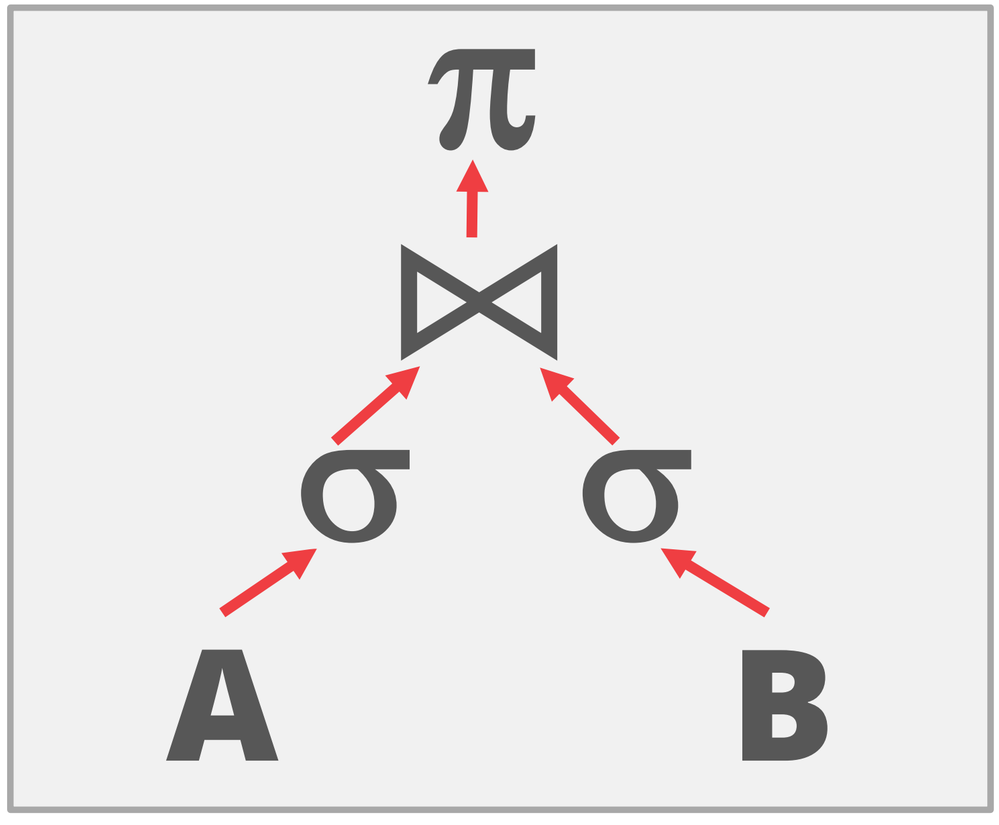


Task Queues

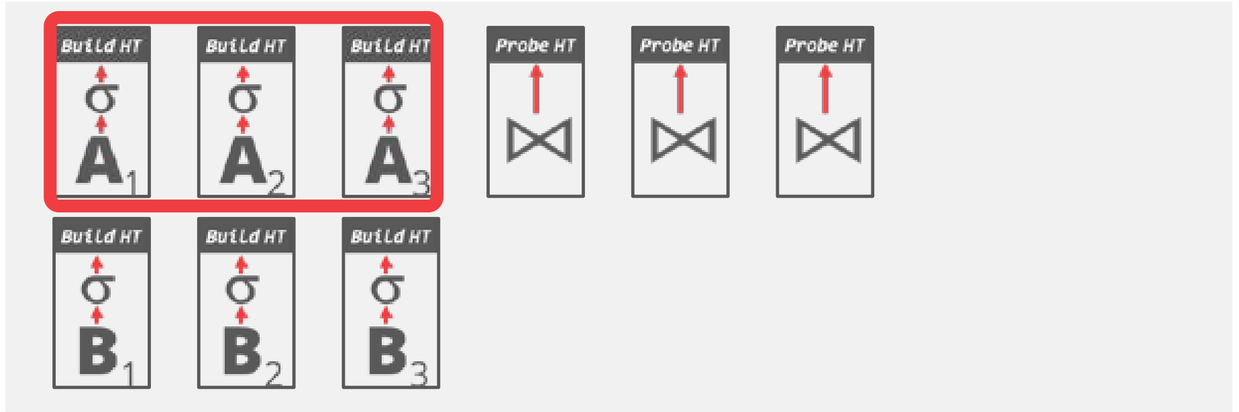


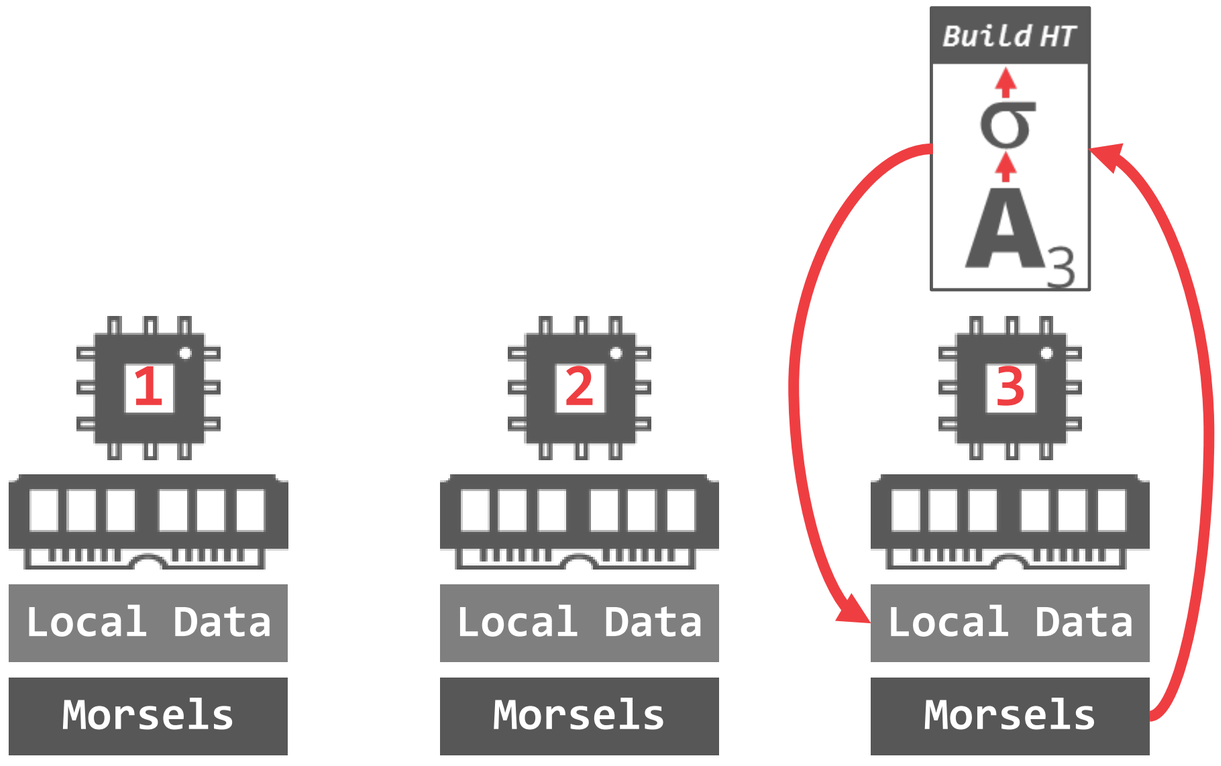
HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



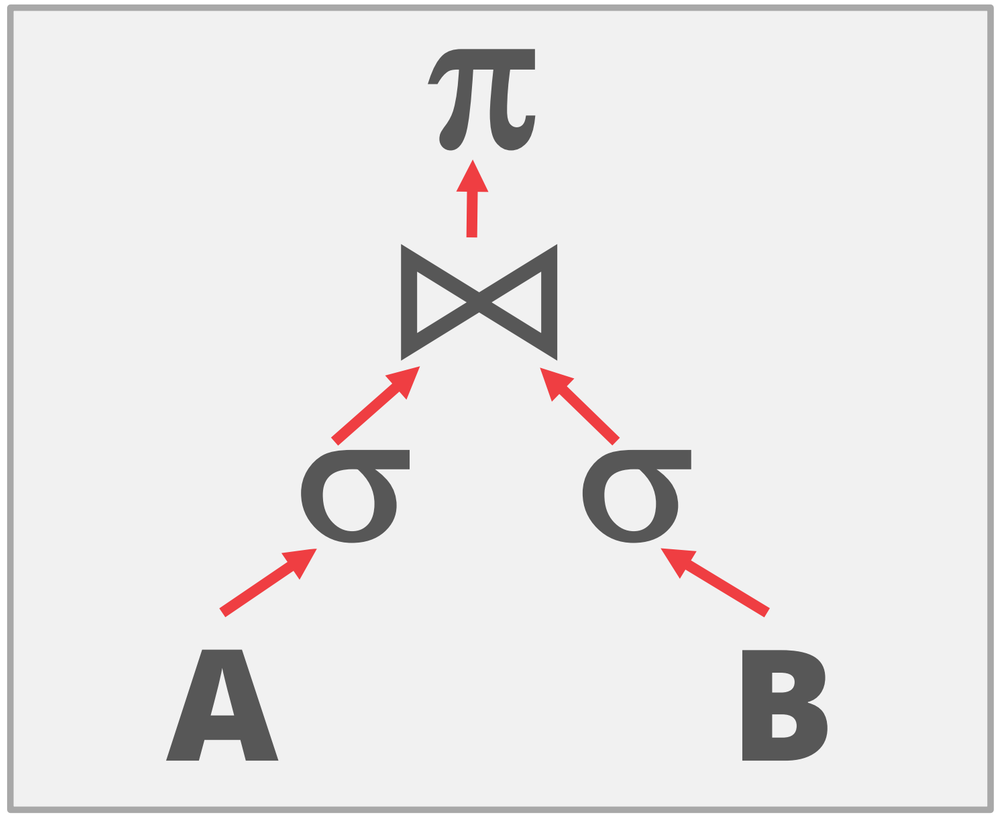
Task Queues



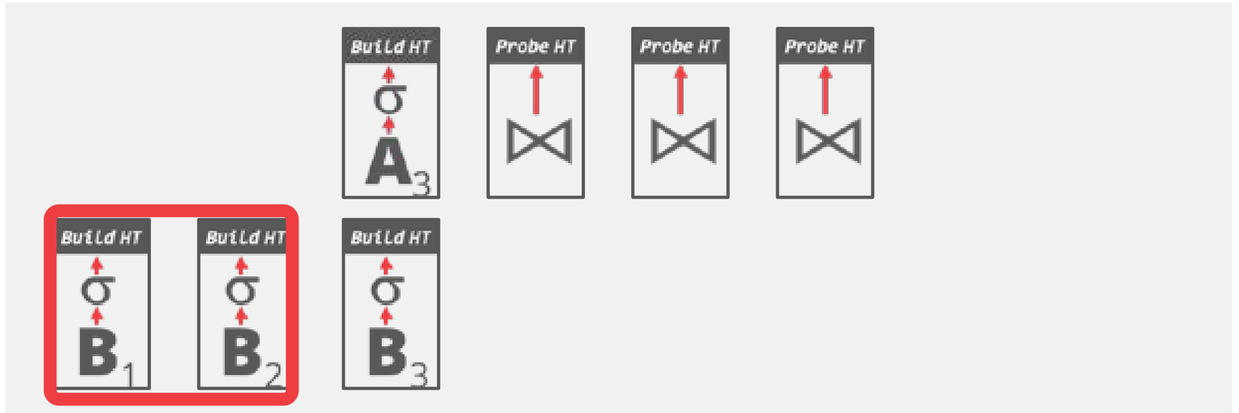


HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

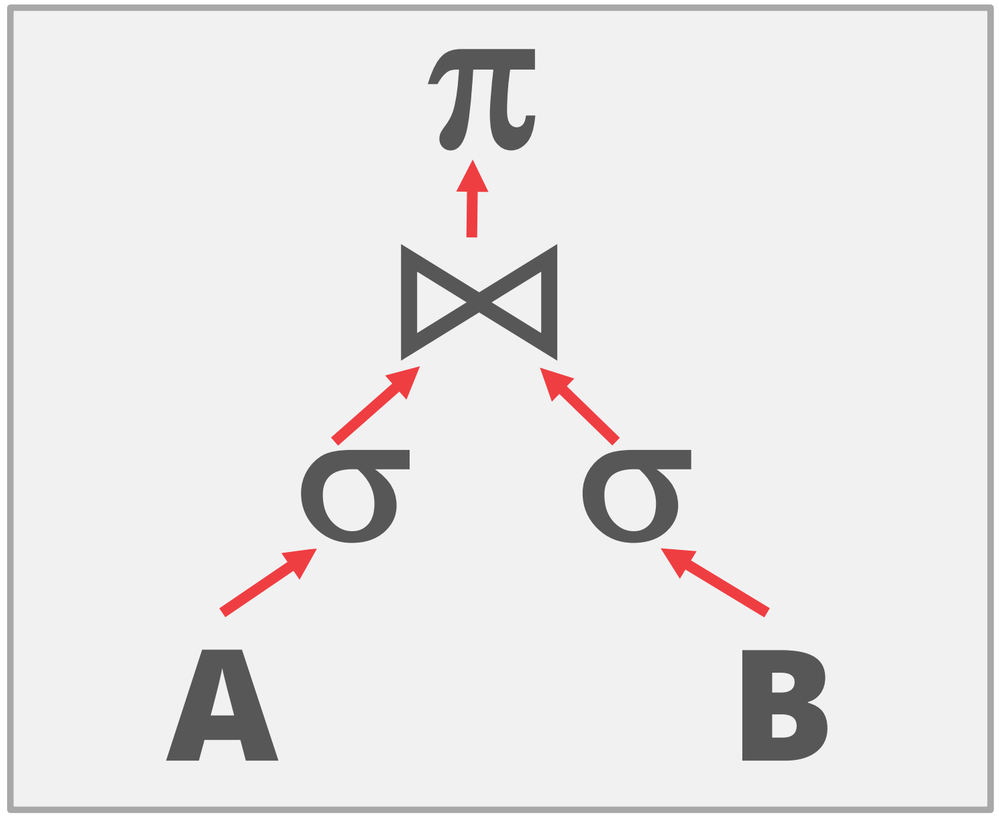


Task Queues

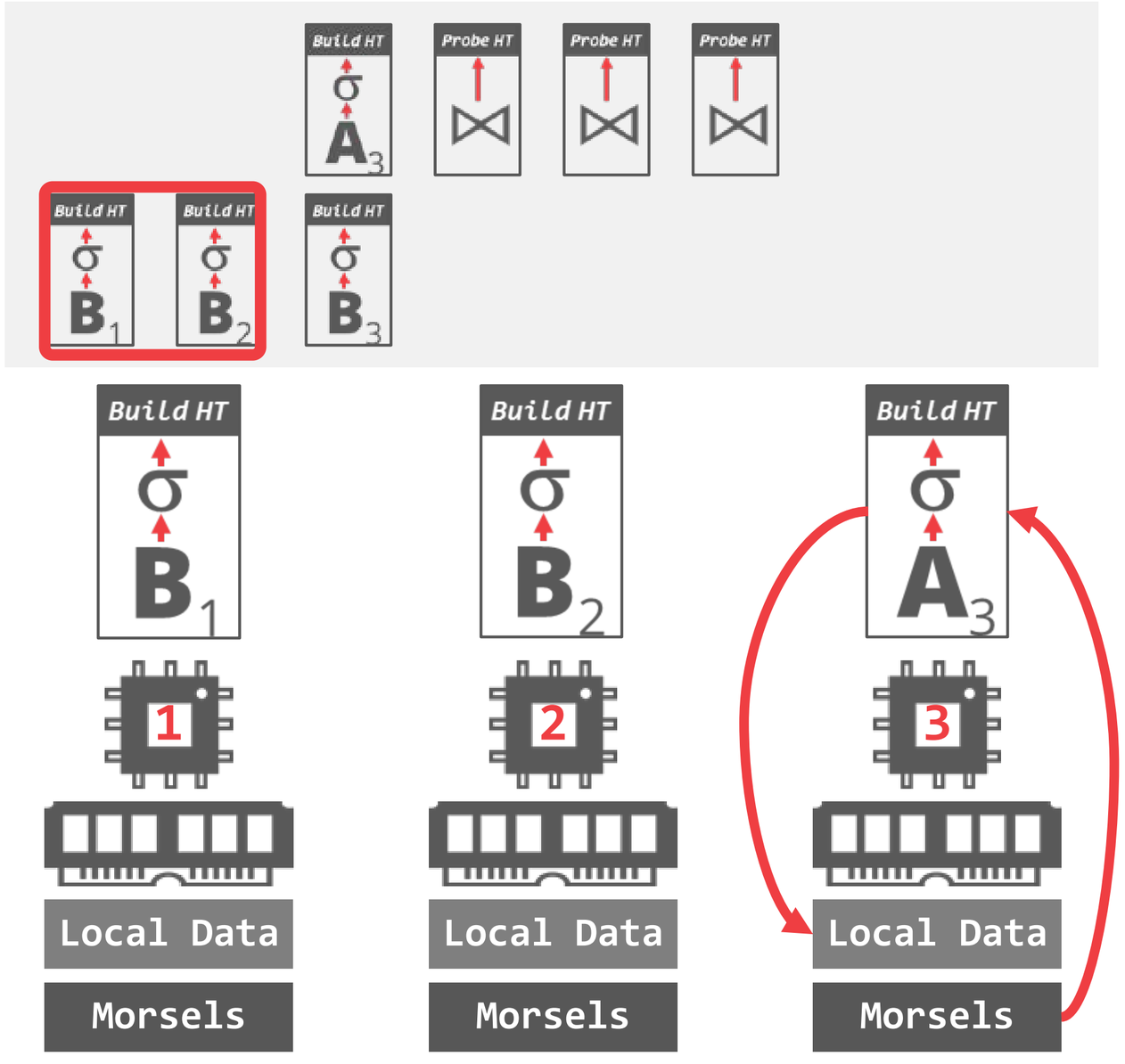


HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

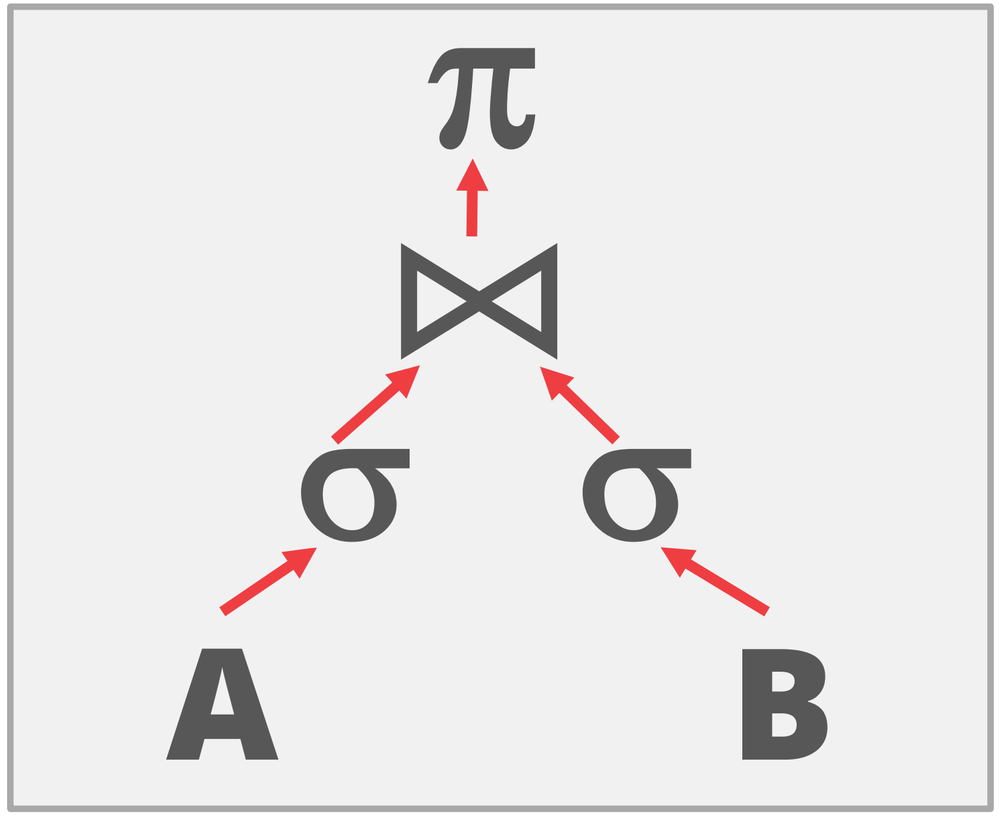


Task Queues

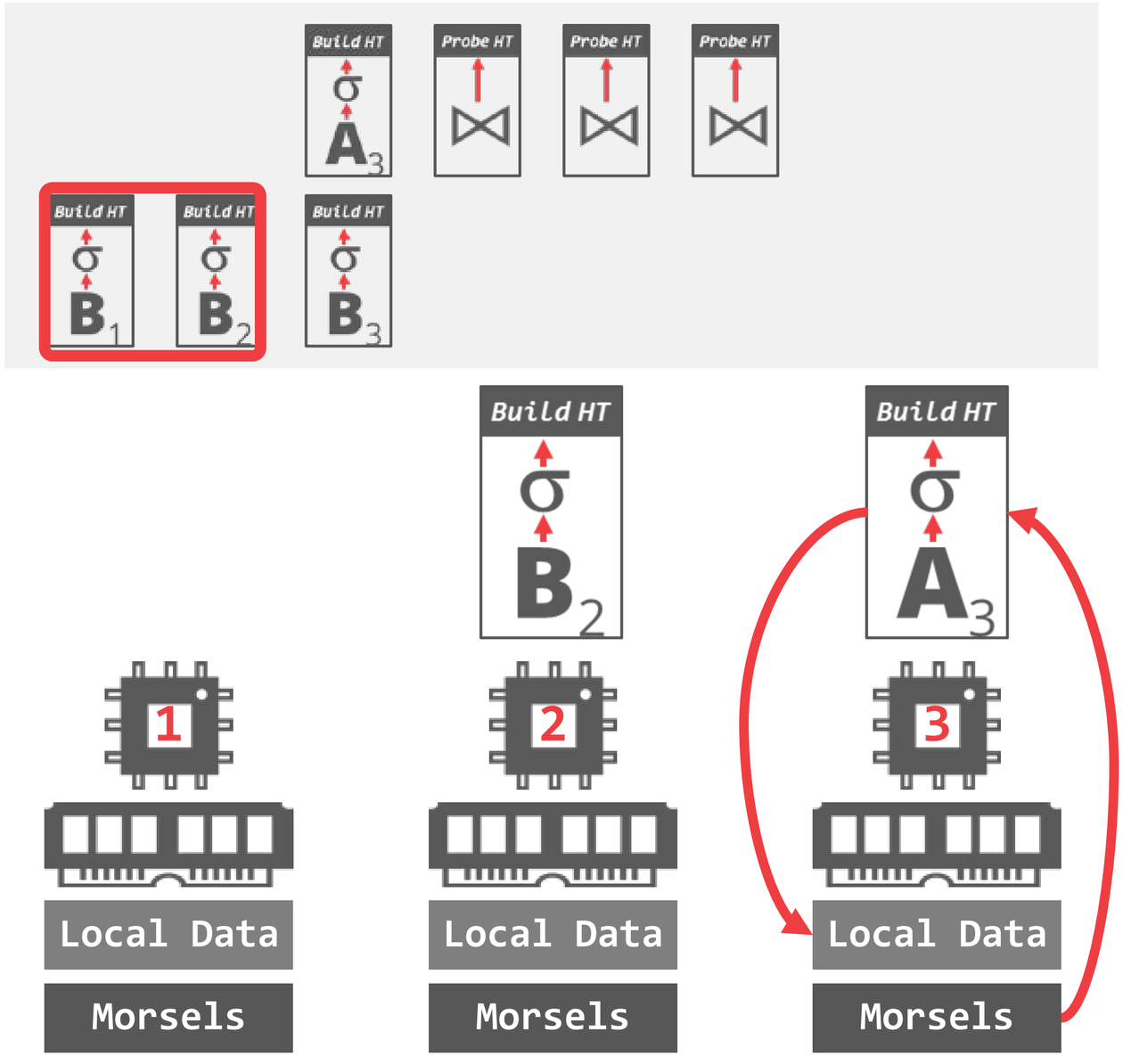


HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

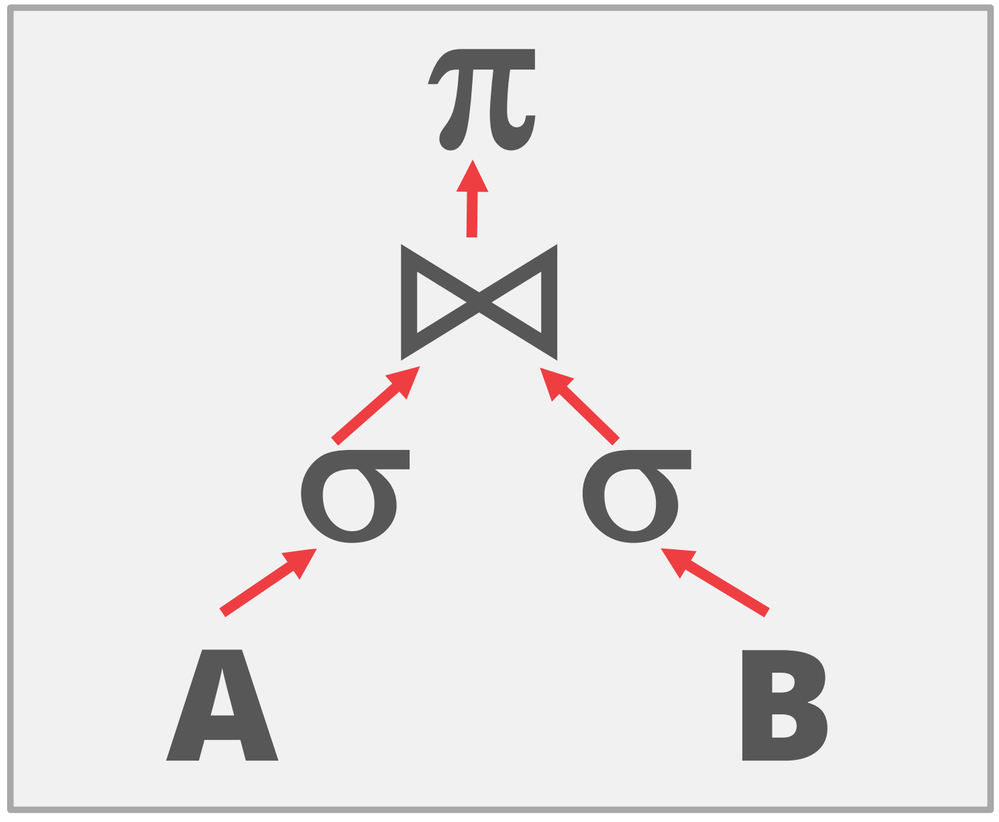


Task Queues

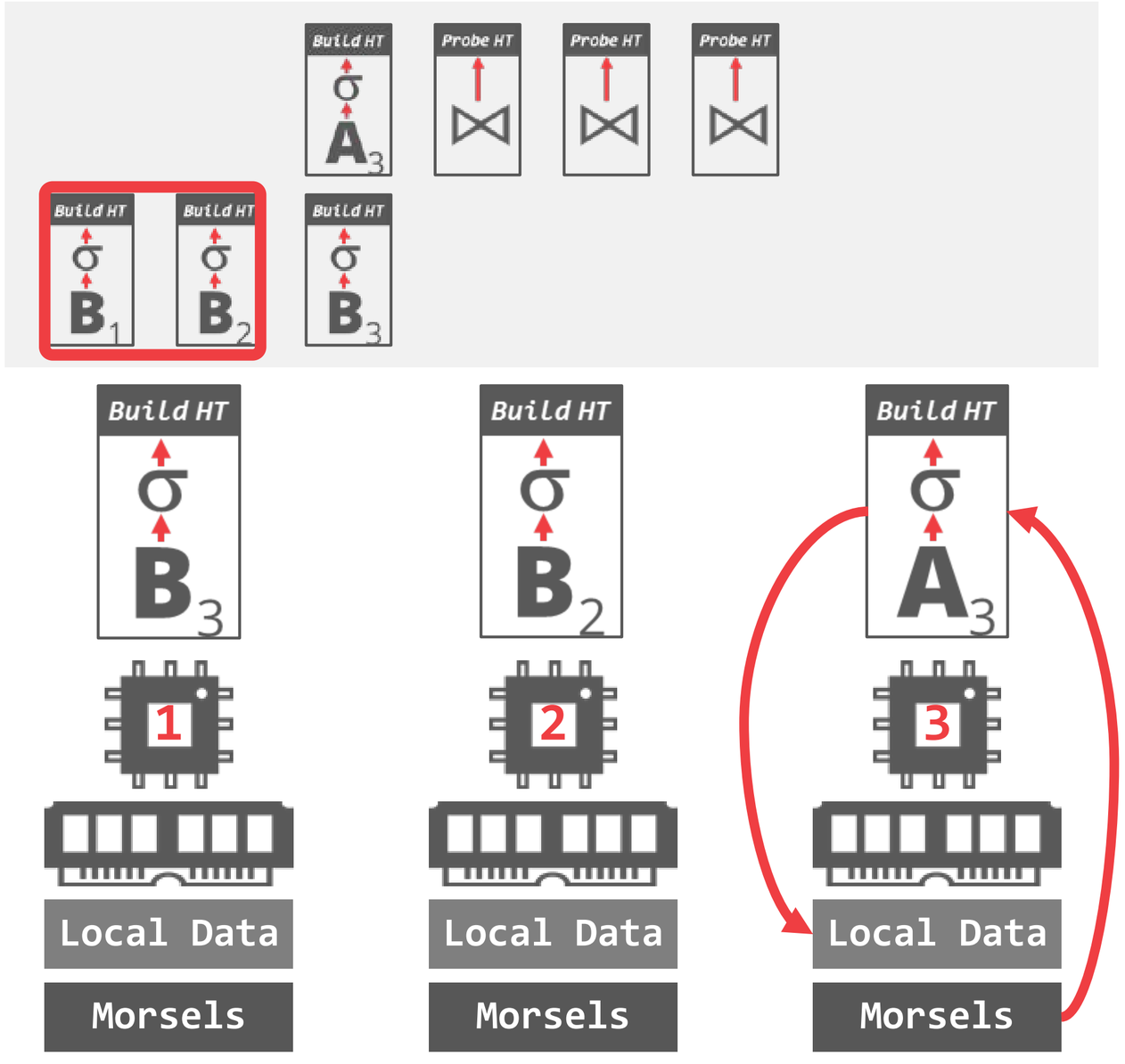


HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

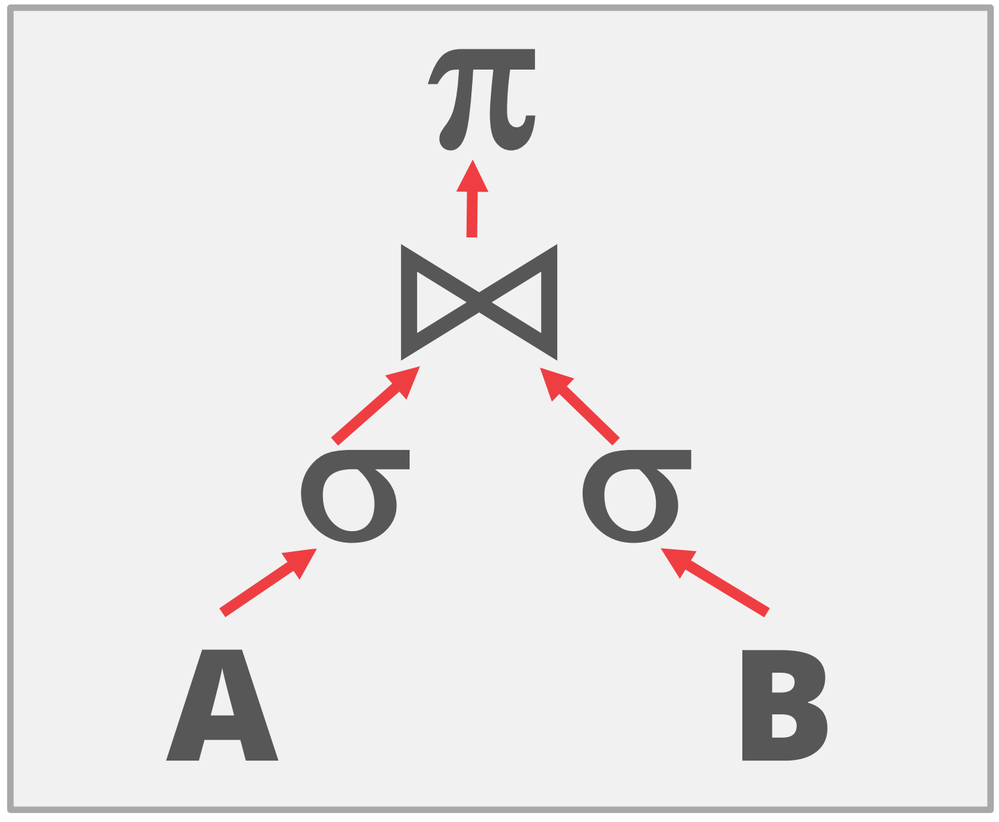


Task Queues

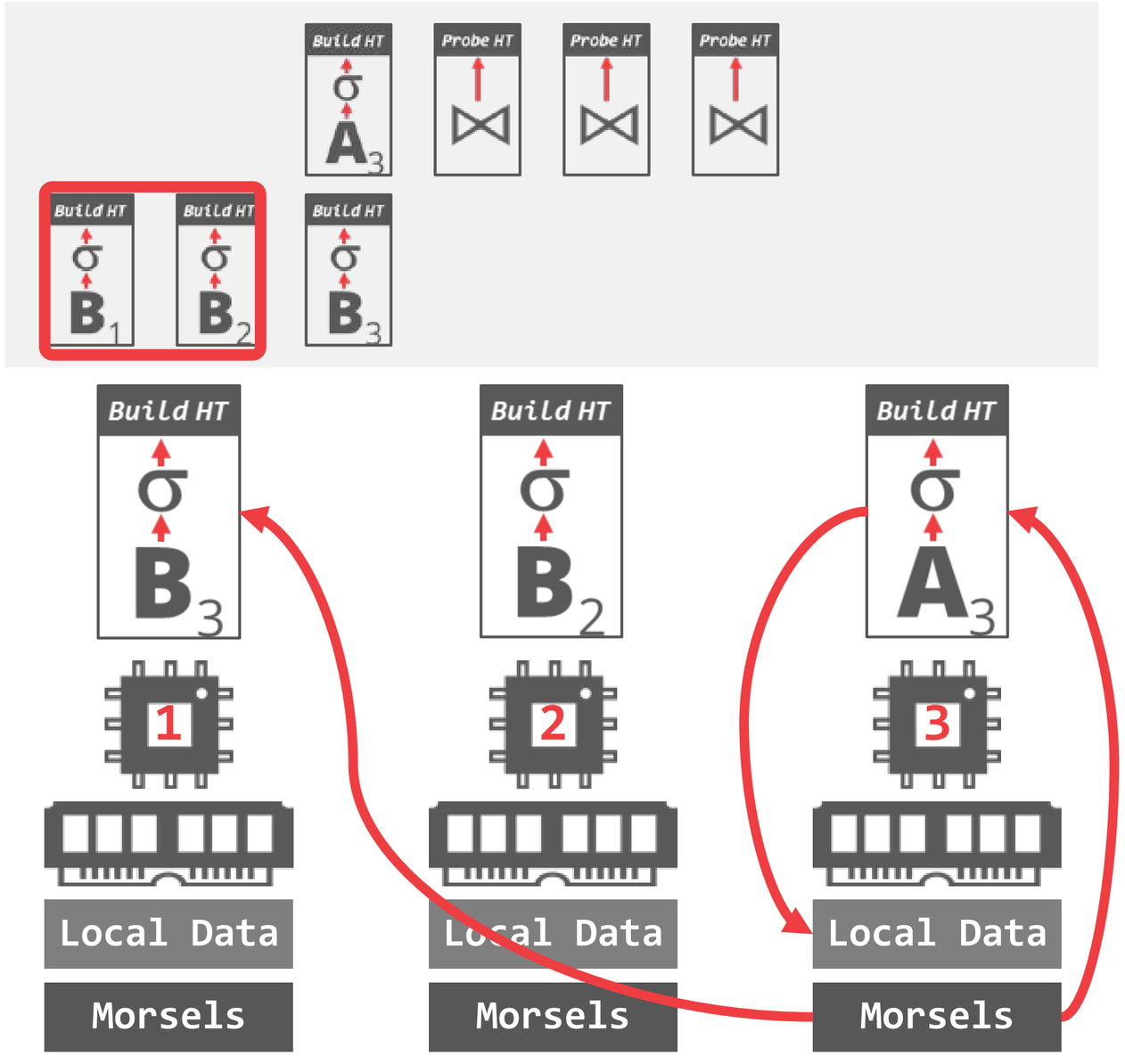


HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

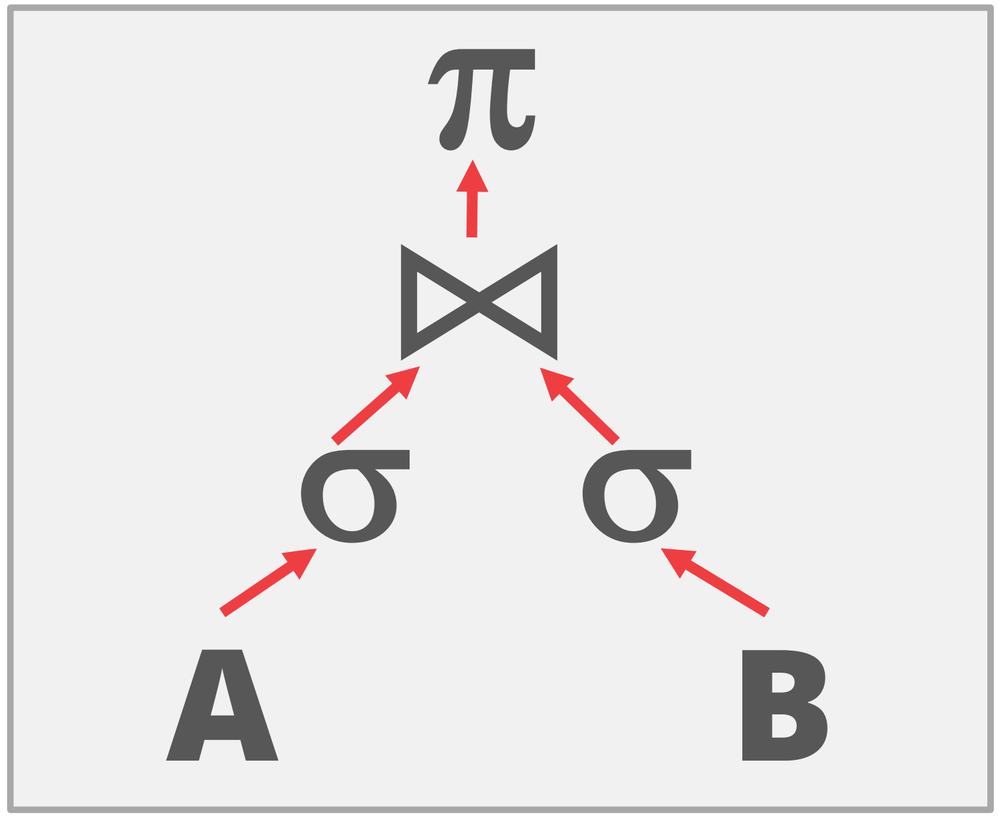


Task Queues

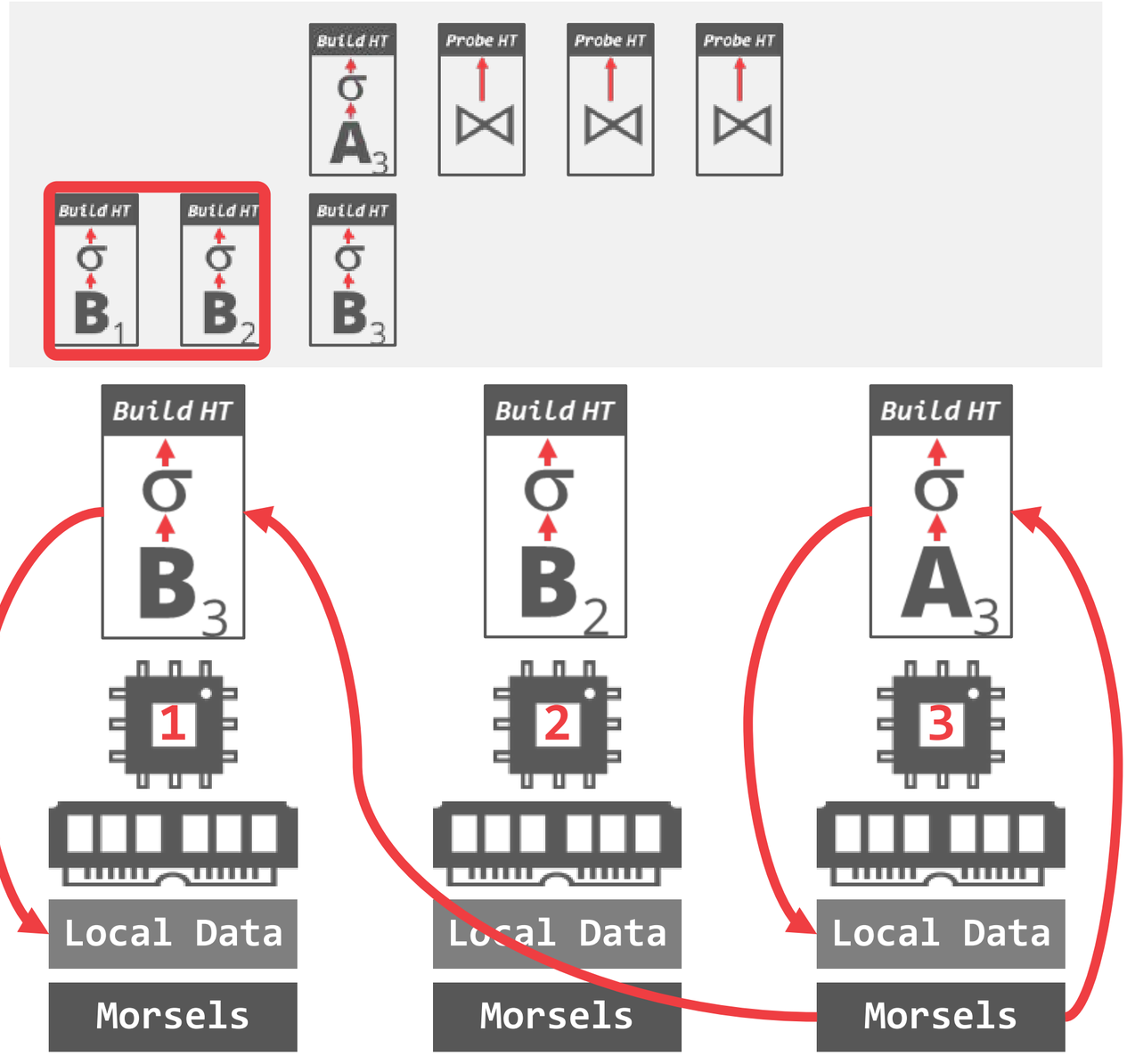


HyPer: Morsel-Driven Execution


```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



Task Queues



NUMA Topologies

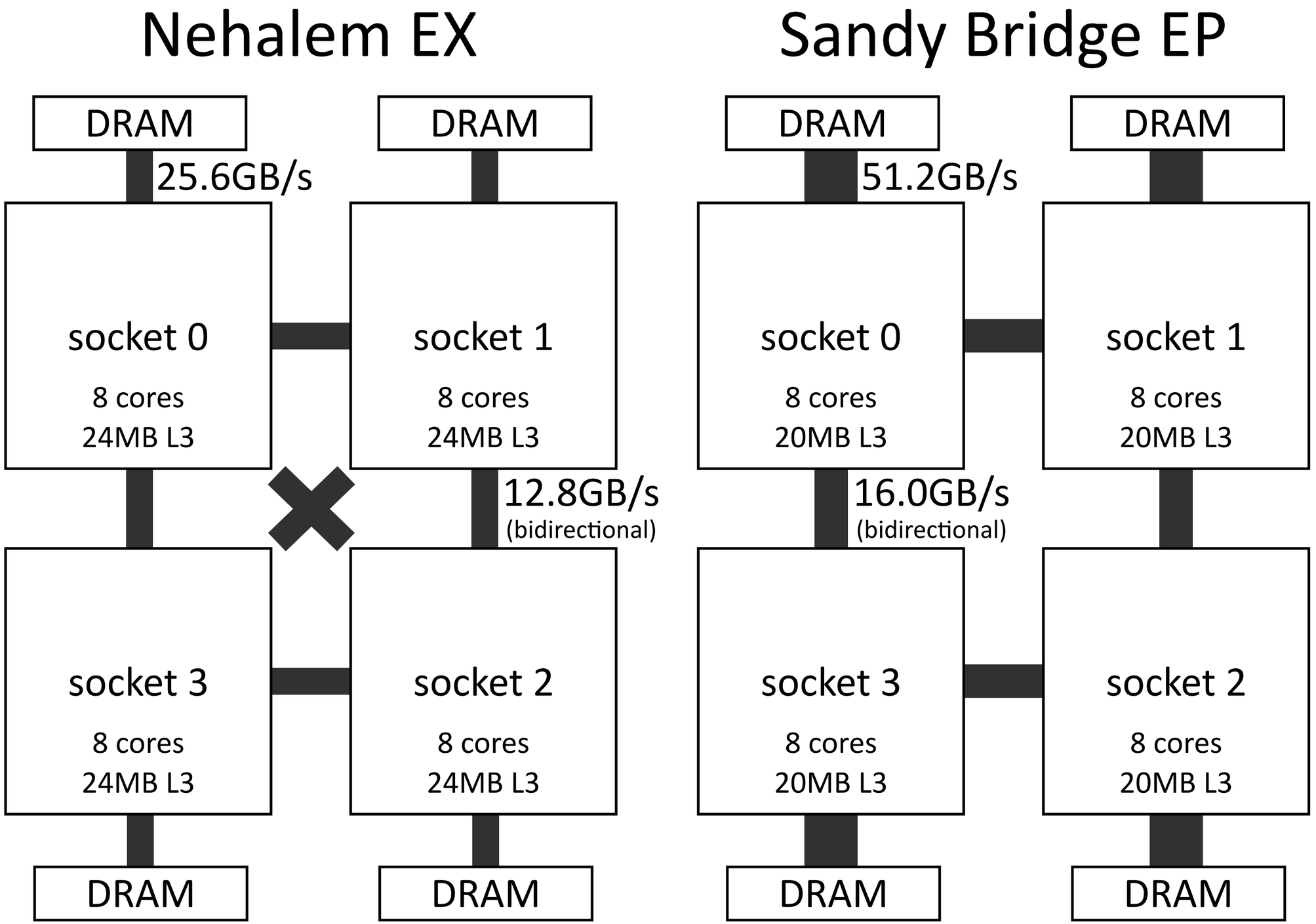
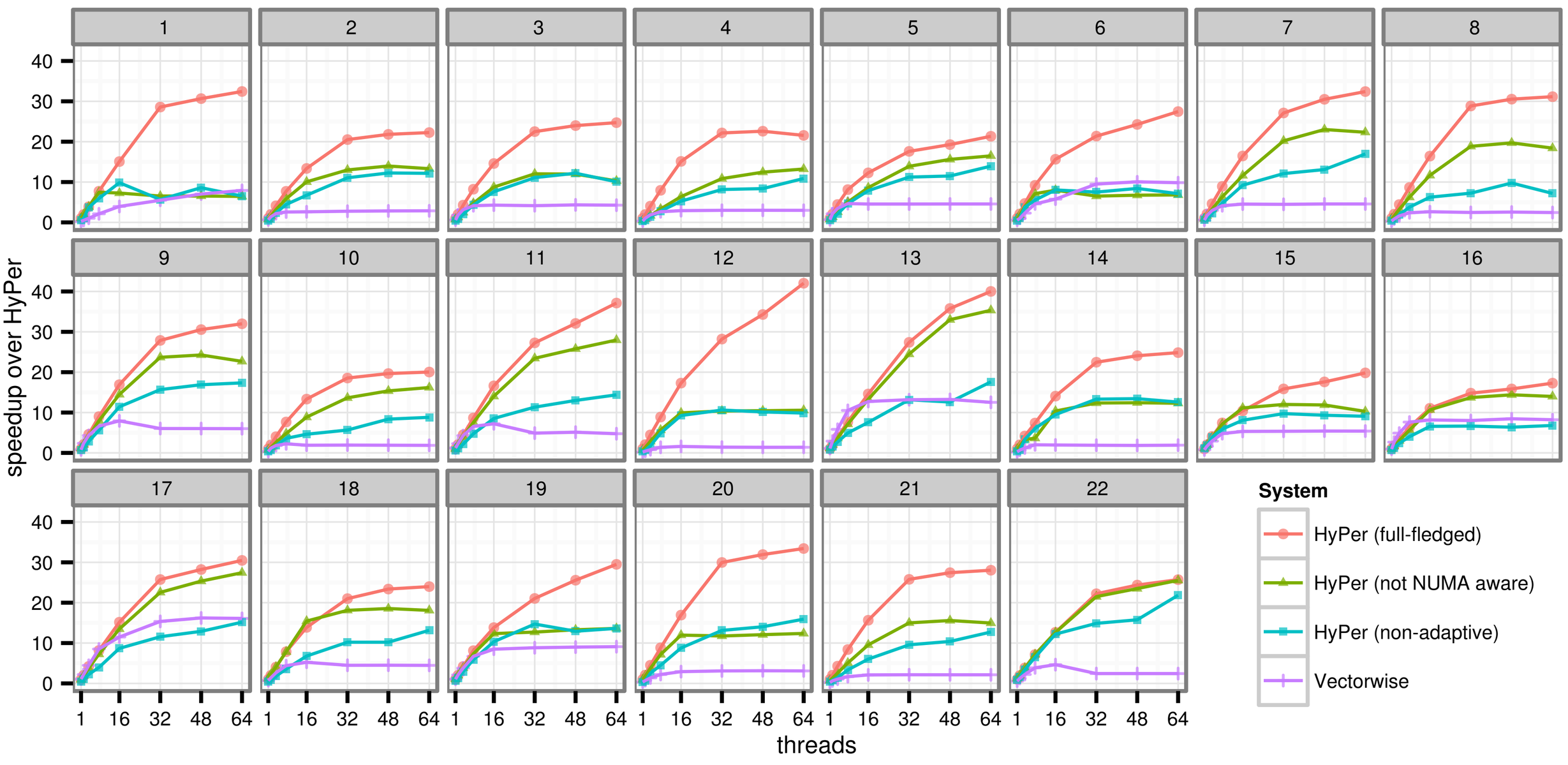


Figure 10: NUMA topologies, theoretical bandwidth

TPC-H Scalability on Nehalem EX

- 4-socket Sandy Bridge EP (Intel Xeon E5-4650L)
- Cores 1 ~ 32 are "*real*", cores 33 ~ 64 are "*virtual*"



Morsel-Driven Scheduling

- Task scheduling for OLAP workloads
  - Partition data by *small morsels (100,000 tuples)* across sockets
  - Pass *morsels* through whole *operator pipelines*
- Because there is *only one worker* per core, they have to use *work stealing* because otherwise threads could sit idle waiting for stragglers.
- Uses a *lock-free hash table* to maintain the global work queues.

- Key Takeaways [Principles]
  - *Fine-grained* scheduling
  - *Full operator* parallelization
  - Low-overhead *synchronization*
  - *NUMA-aware* scheduling

Summary

- Motivation: "Why Parallelism?"
- Amdahl's Law and Gusfafson's Law
- Parallel Programming Models
- Parallelism in In-Memory Databases
- Volcano: Intra-Query Parallelism
- Morsel-Driven Scheduling

thank you
danke
рахмат
謝謝
ngiyabonga
teşekkür ederim
gracias
merci
Баярлалаа
спасибо
faafetai lava
mersi
kia ora
barka
welalin
tack
spas
dank je
misaotra
matondo
paldies
grazzi
mahalo
tapadh leat
vinaka
спасибі
blagodaram
kiitos
dankie
dhanyavad
nanni
nandri
maururu
hvala
bedankt
bayarlalaa
gràçie
köszönöm
akun
dankon
aćiū
хвала
asante
manana
obrigada
tenki
murakoze
chokrane
djiere dieuf
tau
mochchakkeram
дякую
mamnun
enkosi
dziękuję
chnorakaloutioun
gratias ago
gràcies
sulpáy
go raibh maith agat
sobodi
dėkuji
obrigado
sagolun
sukriya
kop khun krap
taiku
arigatô
takk
dakujem
trugarez
mèsi
didi madloba
ありがとう
tanemirt
grazie
najis tuke
terima kasih
rahmet
kam sah hamnida
rahmat
diolch
dhanyavadagalu
shukriya
mercê
мерси
তোমাকে ধন্যবাদ
감사합니다
xièxie
ευχαριστώ