Sub-Operators in Query Execution with a focus on Joining and Filtering

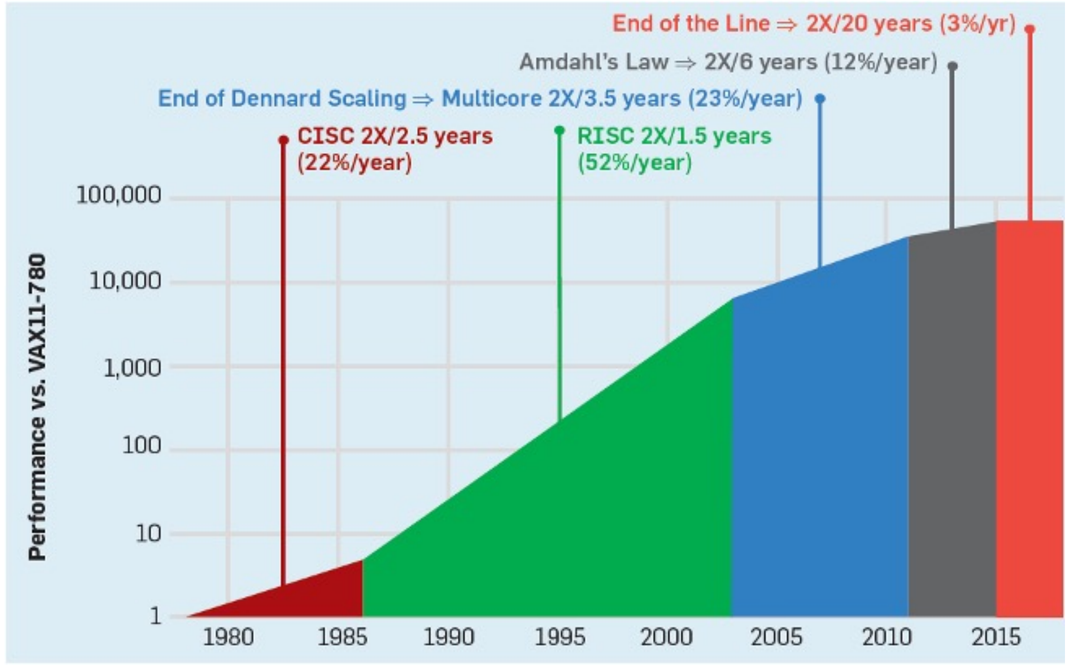
Maximilian Bandle

Technische Universität München
FG DB 2022

Why can’t we just keep going as usual?

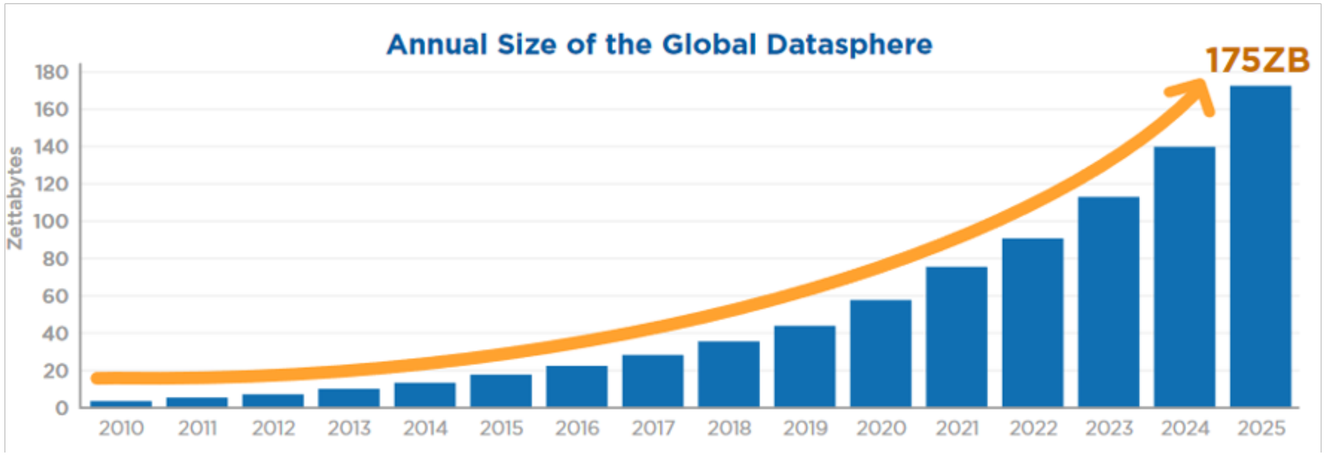
CPU Performance vs Data Expanse

End of Dennard Scaling



John Hennessy and David Patterson Deliver Turing Lecture at ISCA 2018

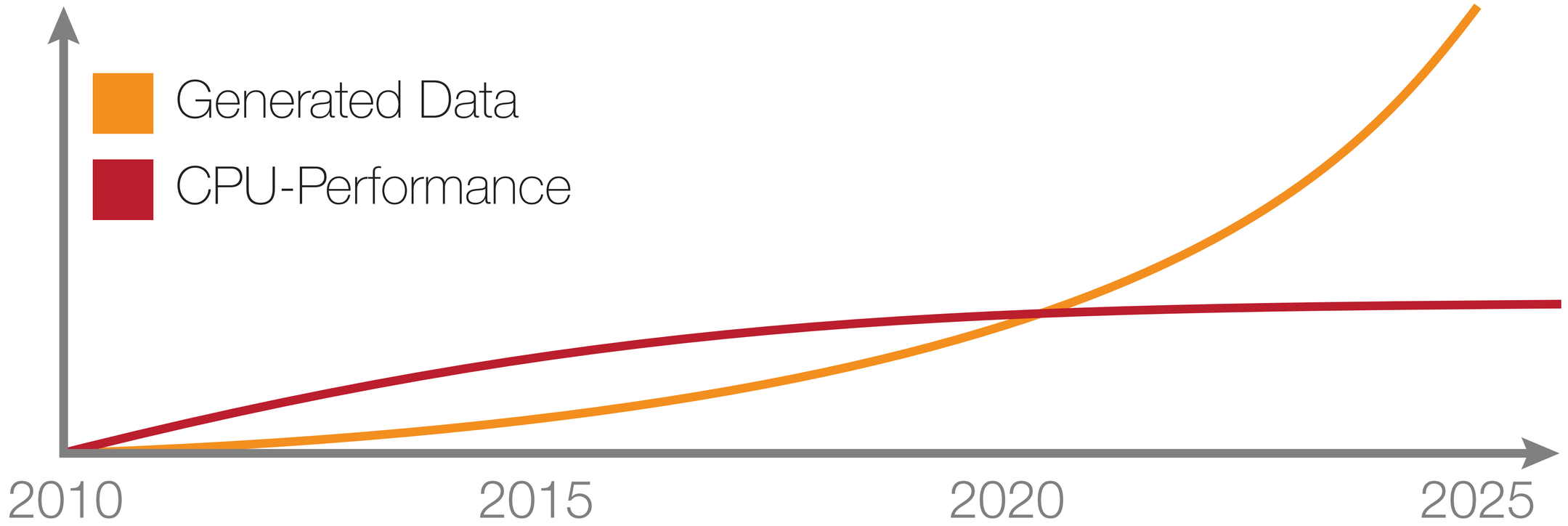
Increasing Data Generation



Data Age 2025,IDC Global DataSphere, May 2020 (sponsored by Seagate)

Data Expanse vs CPU Performance

The gap increases

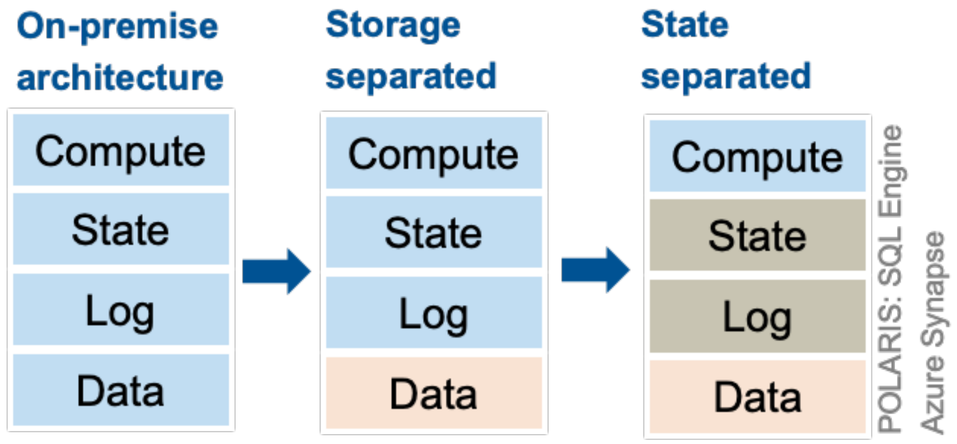


Getting all data to compute is getting challenging

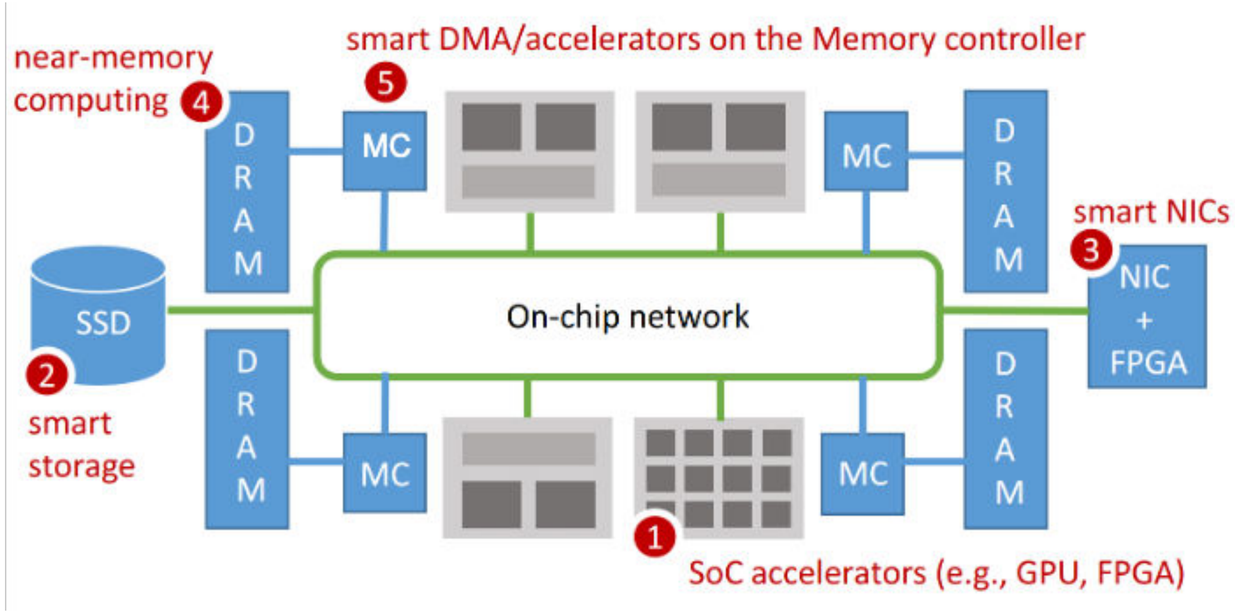
Cloud Evolution impacts our Decisions

Separation of Data and computation

Trend to separate Compute & Data



Accelerators & Disaggregated HW



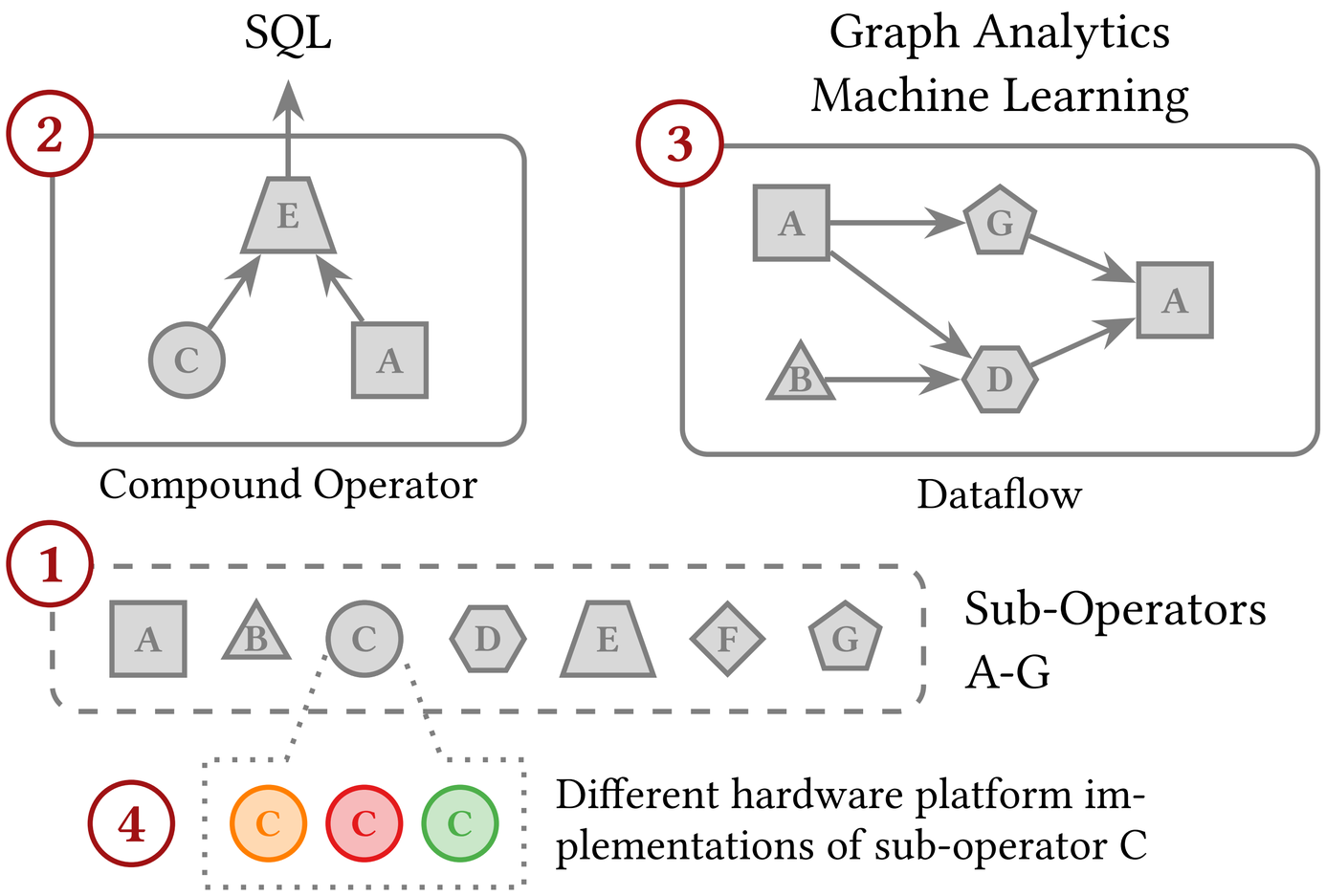
Jana Giceva: Building Cloud-native Data Systems for the Post-Moore Era

=> Cloud Vendors are driving hardware innovation

How should a cloud-native data system for the post-Moore era look like?

Make sub-operators first class citizens

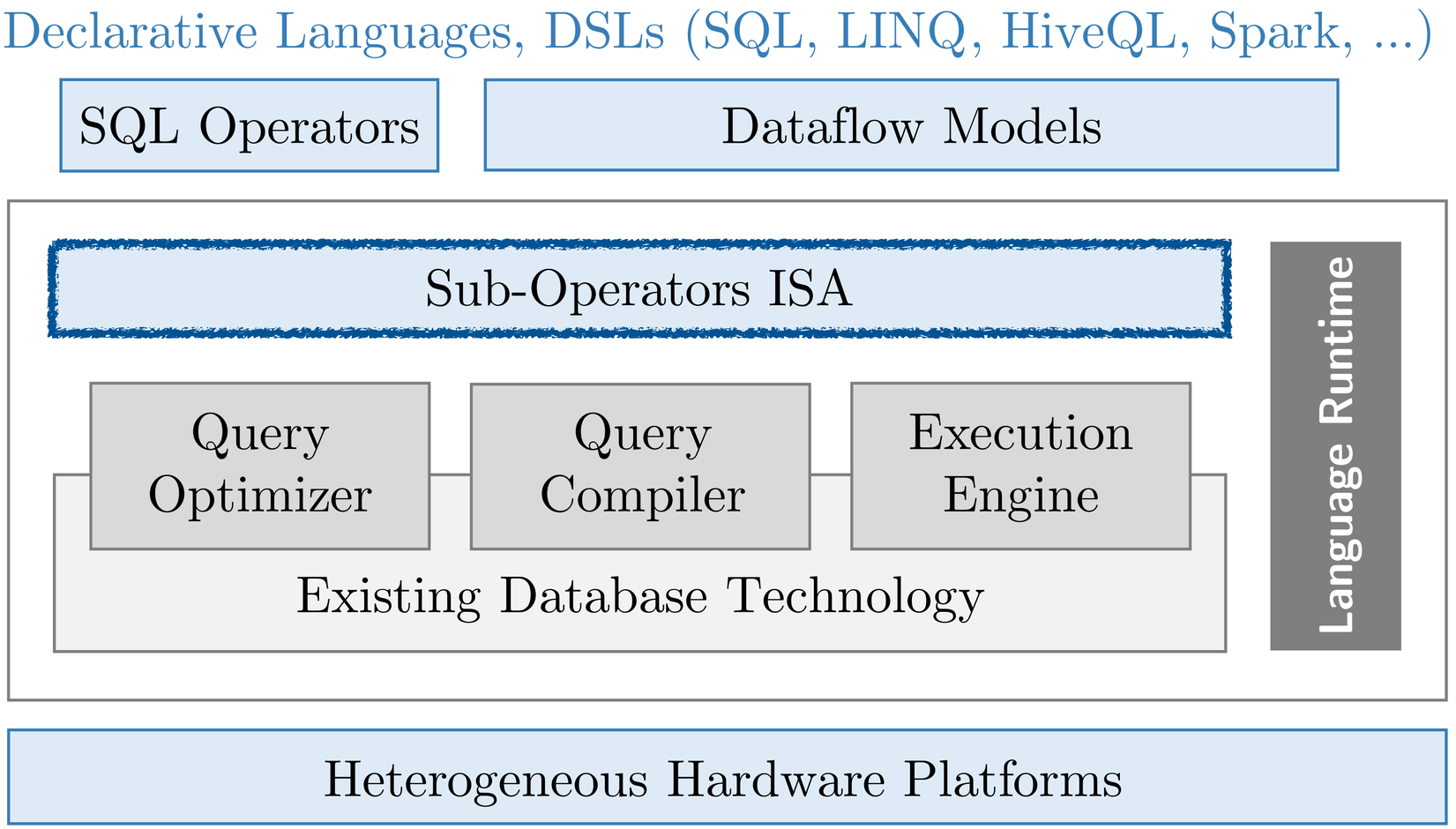
Split relational operators into more flexible sub-operators



1. Logical functions that perform basic data transformation and management tasks.
2. Construct classic relational algebra operators for SQL when combining.
3. Flexible for composing to map to arbitrary other dataflows.
4. Granularity allows efficient compilation to heterogenous hardware and offloading to where data sits or moves.

Sub-Operator based system architecture

Transform the DBMS into an executor for the sub-operator ISA



Query Optimizer

- What pays off to be offloaded?
- Capture the device's capabilities.

Query Compiler

- Codegen the accelerator's ISA

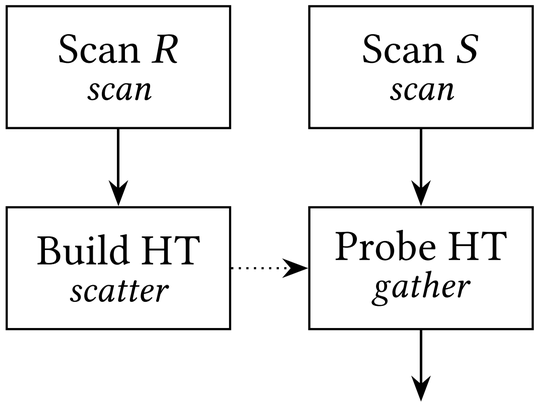
Execution engine

- Co-design with device drivers
- Co-design with data-center scheduler

Modularis: Modular Data Analytics for Hardware, Software, and Platform Heterogeneity
Building Advanced SQL Analytics From Low-Level Plan Operators (SIGMOD 2021)
Charting the design space of query execution using VOILA (VLDB 2021)

Build complex dataflows from simple components

Use simple operators to build a hash join



Example Categories for Sub-Operator ISA Operations

- **Sequential Access**

  Materialize, Scan
- **Random Access**

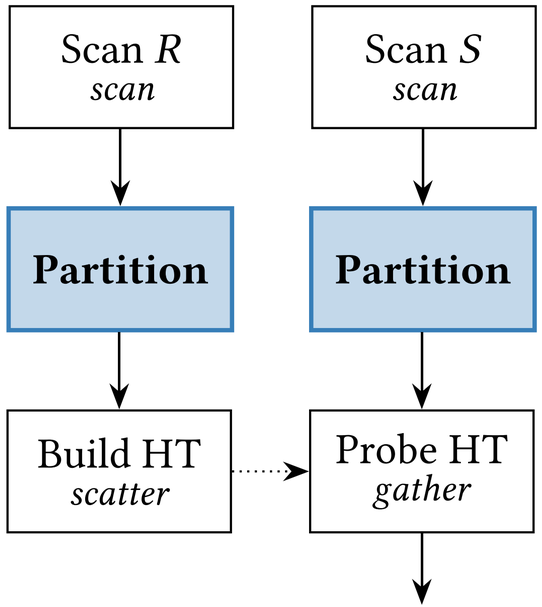
  Scatter, Gather
- **Compute**

  Map, Fold
- **Control Flow**

  Loop

Build complex dataflows from simple components

Use simple operators to build a hash join



Example Categories for

Sub-Operator ISA Operations

- **Sequential Access**

  Materialize, Scan
- **Random Access**

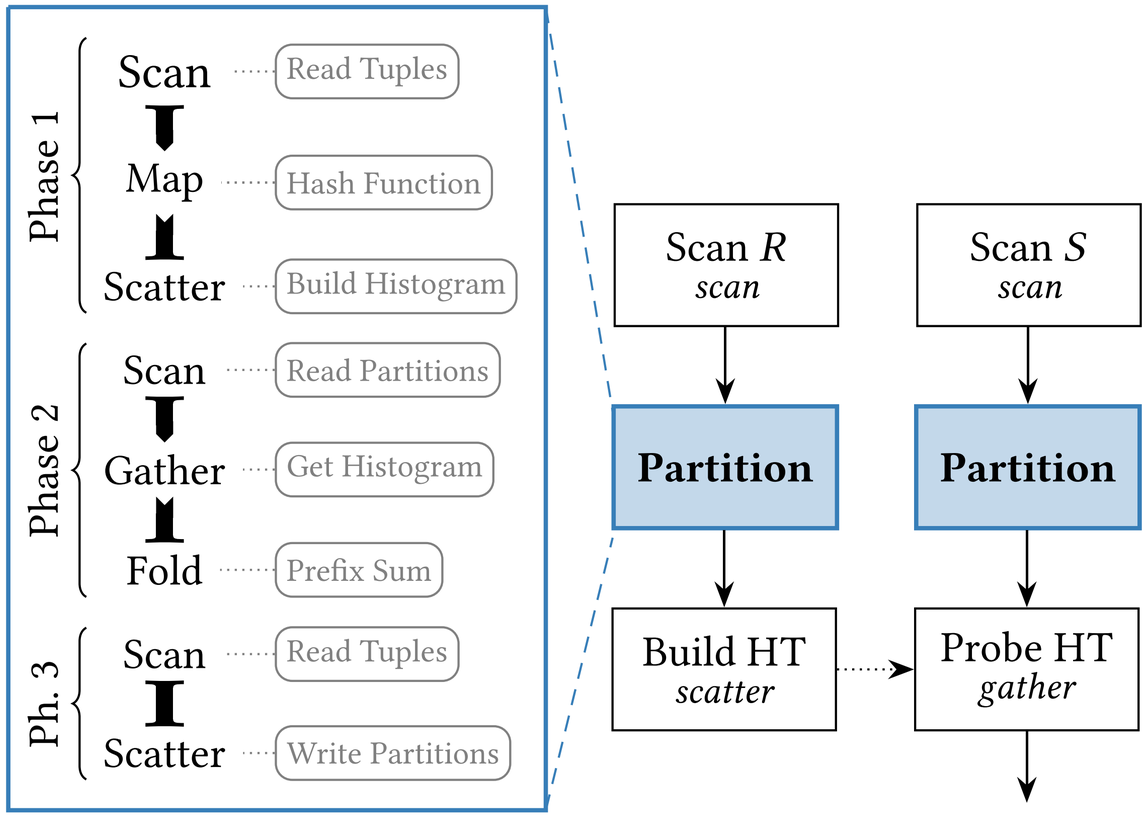
  Scatter, Gather
- **Compute**

  Map, Fold
- **Control Flow**

  Loop

Build complex dataflows from simple components

Use simple operators to build a hash join



Example Categories for Sub-Operator ISA Operations

- **Sequential Access**

  Materialize, Scan
- **Random Access**

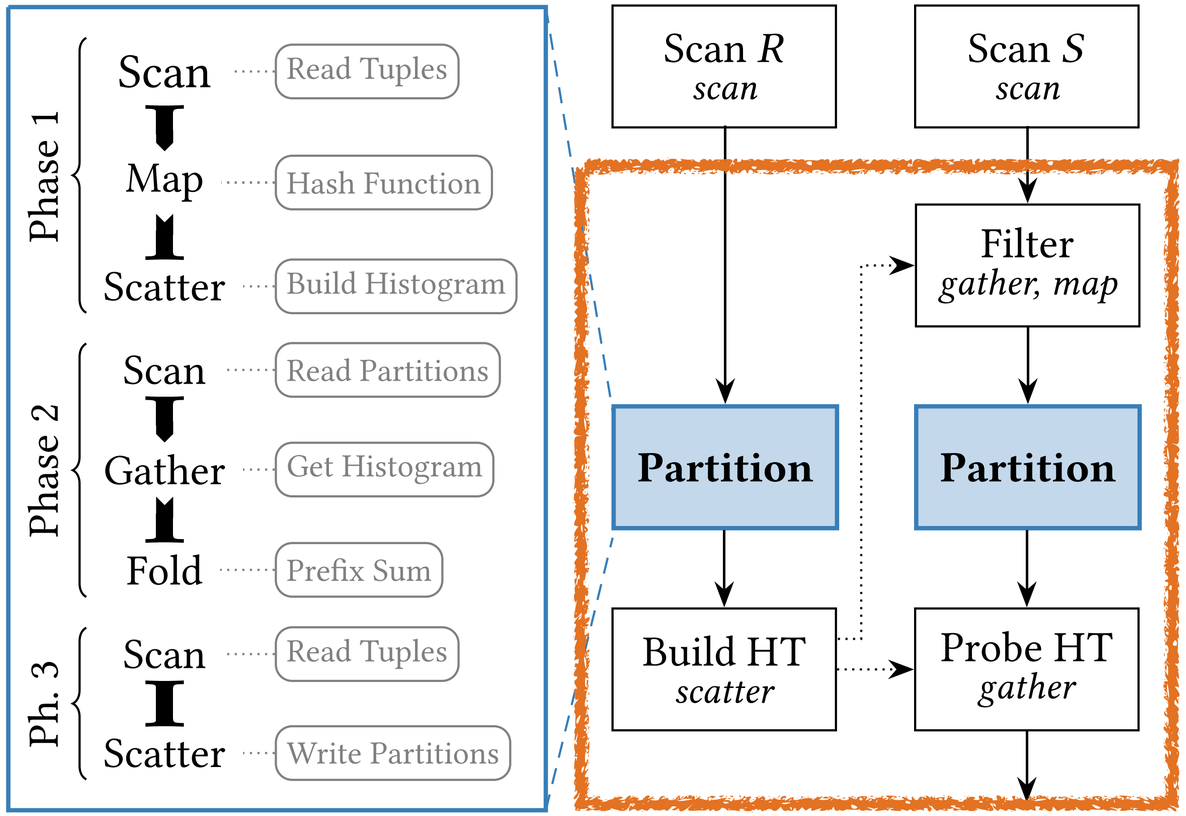
  Scatter, Gather
- **Compute**

  Map, Fold
- **Control Flow**

  Loop

Build complex dataflows from simple components

Use simple operators to build a hash join



Example Categories for

Sub-Operator ISA Operations

- **Sequential Access**

  Materialize, Scan
- **Random Access**

  Scatter, Gather
- **Compute**

  Map, Fold
- **Control Flow**

  Loop

General Purpose Radix Join Implementation

Compare Hash Joins Variants using TPC-H

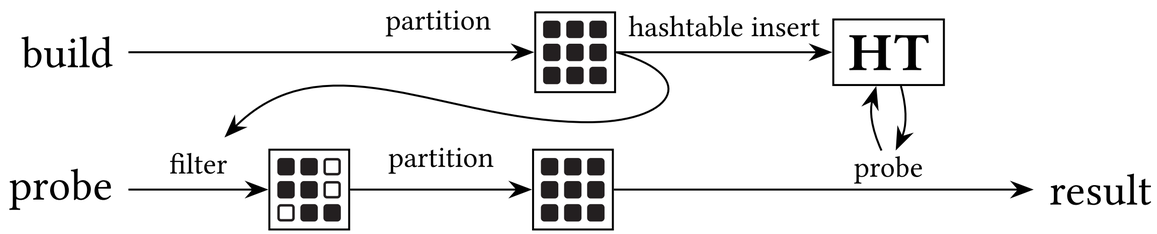
UMBRA

Code Generating Database

Morsel-Driven Execution
Efficient non-partitioned Hash Join

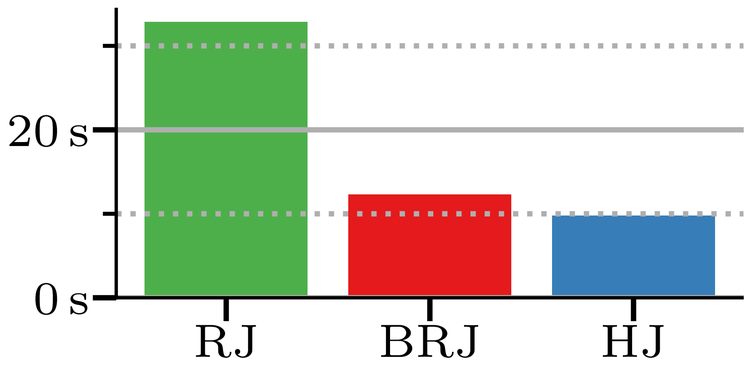
Microbenchmarks

Radix Partitioned Join



TPC-H SF100

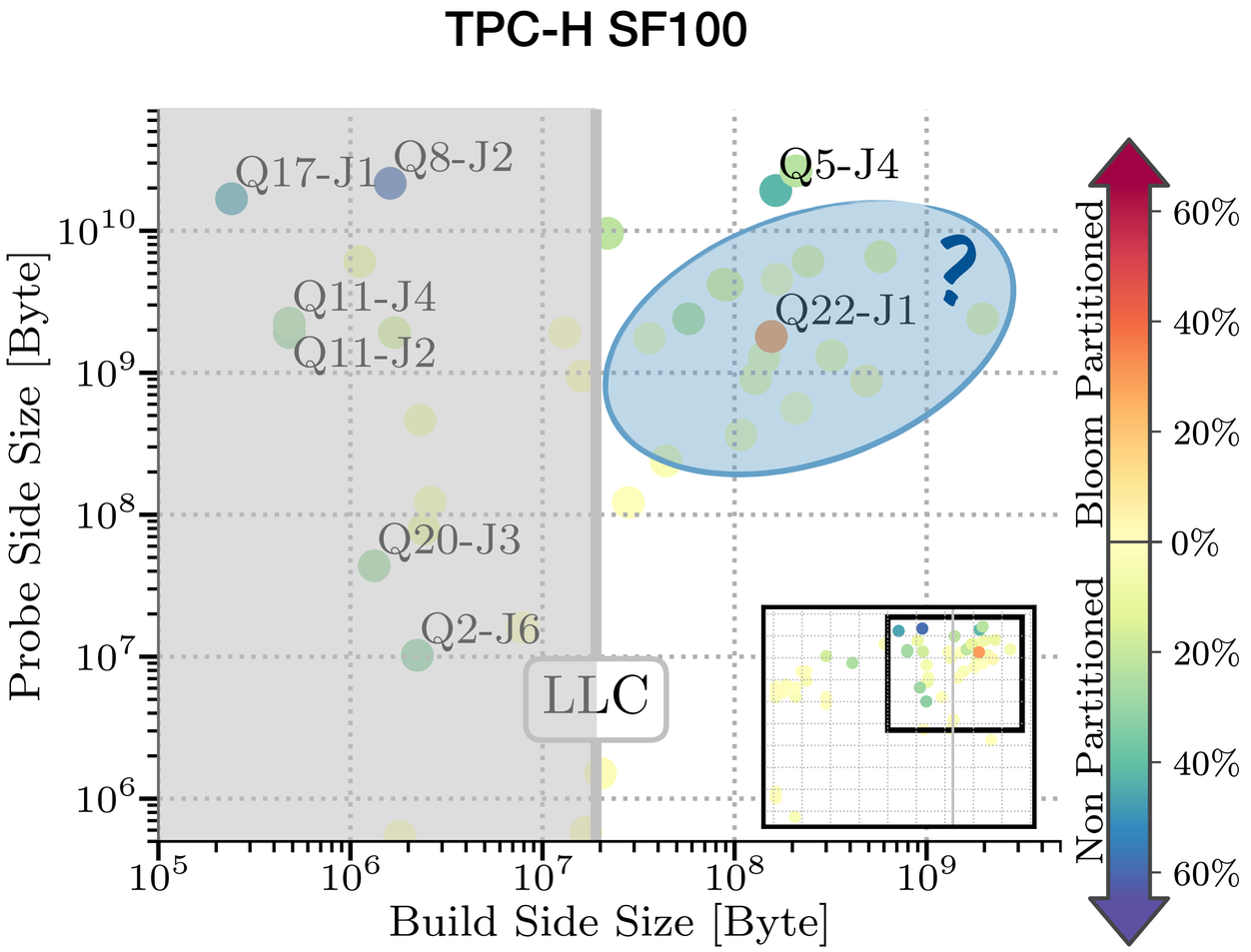
(w/o Q8, Q9, Q21)



Bloom Filters already used by: Barber et al. – Memory-efficient hash joins

Performance Critical Factors for Hash Joins

When does radix partitioning pay off?



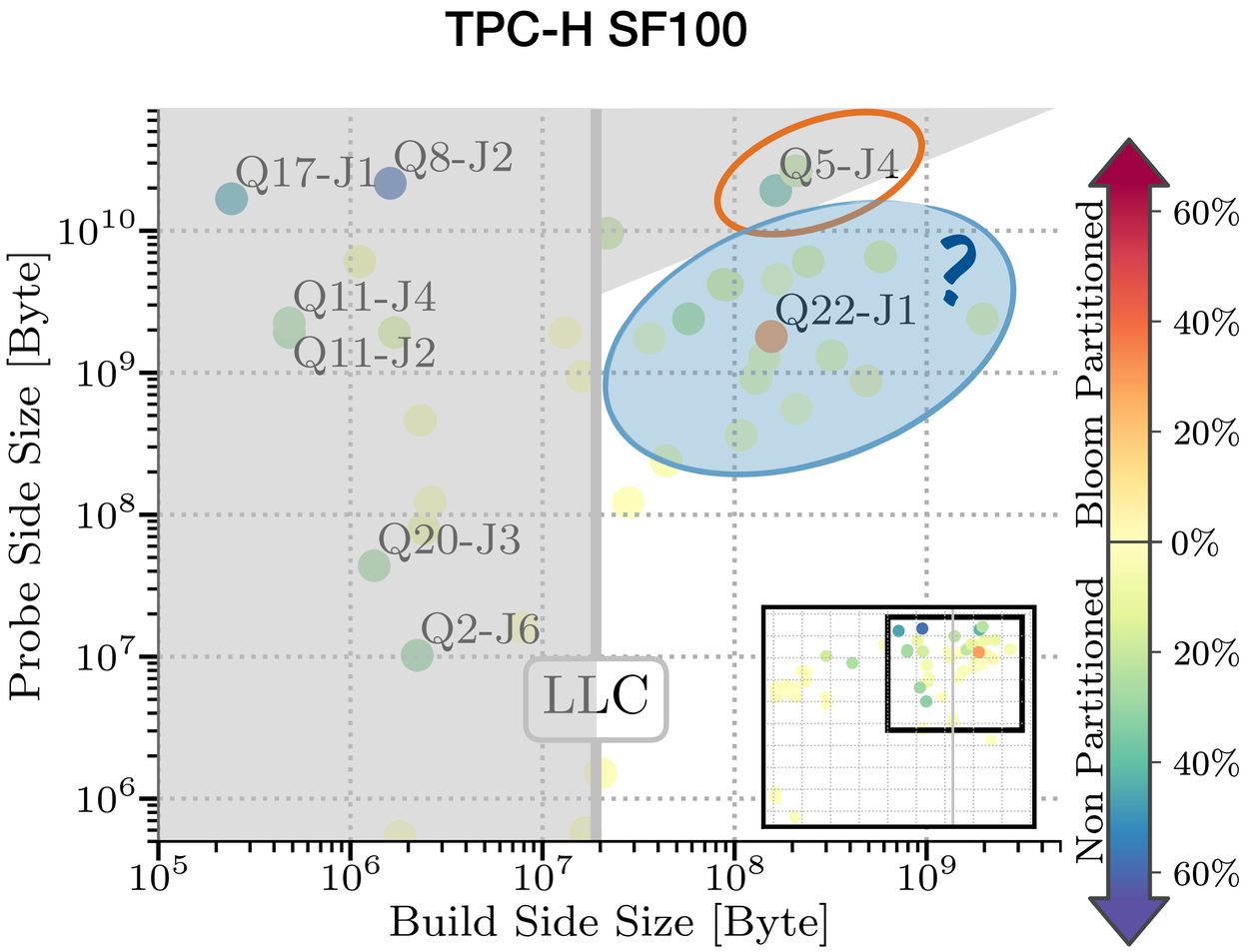
- As expected, small joins don't need partitioning
- But, a large build side does not always profit from partitioning

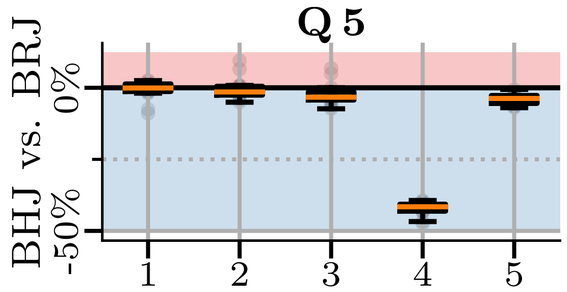
- In fact, only one join in one query benefits from using Bloom Radix Partitioned Join

When does radix partitioning pay off?

Size Difference between Build and Probe Side

Detailed TPC-H Analysis with Query 5 as example





- Single Join dominates the runtime difference
- Build side is larger than the LLC
- Probe side is 100x bigger build side size

- Large Partitioning Overhead
- Sizes should be the same order of magnitude

When does radix partitioning pay off?

Should you use partitioning in a real system?

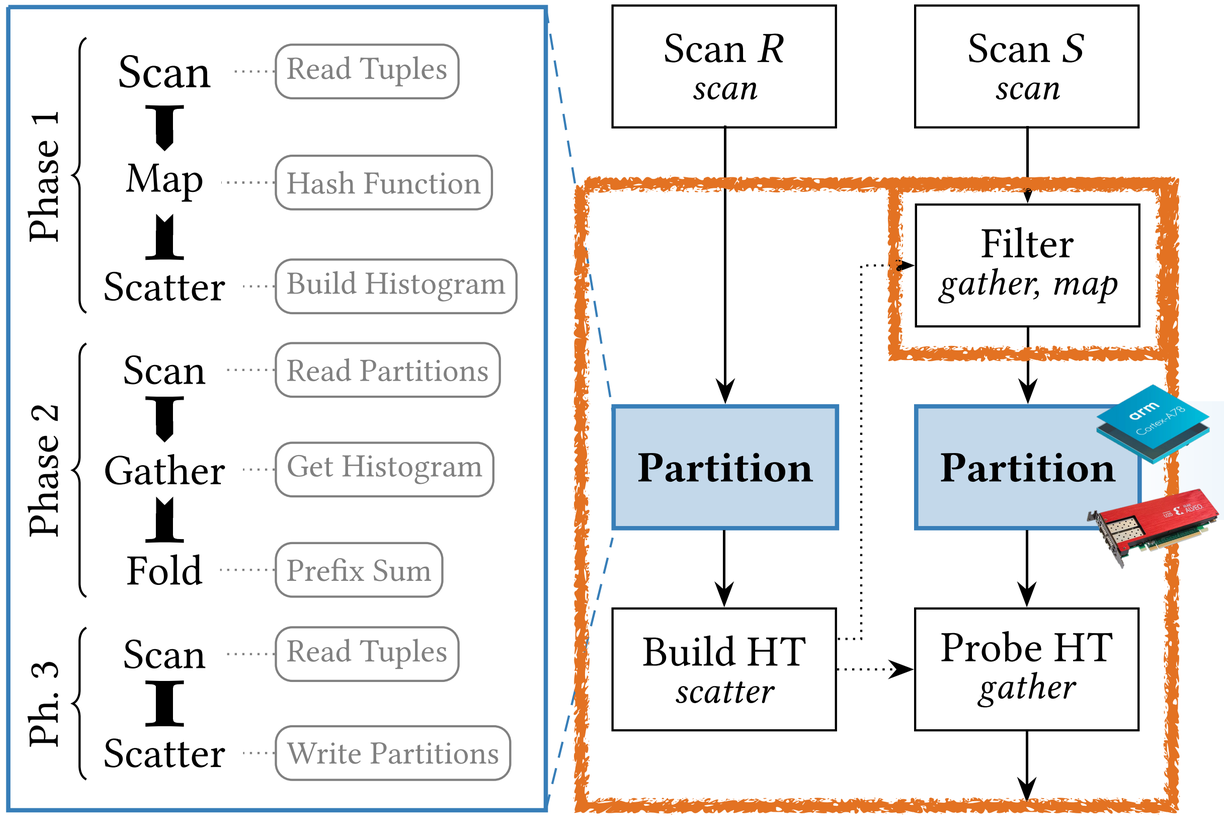
- **Several factors** decide when partitioning pays off
- **Radix-Partitioned joins** are **very sensitive** to any deviation from near-optimal workload characteristic
- **Non-partitioned joins** perform **better** in TPC-H and most realistic workloads

Suitable workloads for partitioned join

Factors	Workable	Beneficial
Selectivity	handled by Bloom filter	
Build Size	$> LLC$	$\gg LLC$
Size Difference	$< \times 50$	$< \times 10$
Payload Size	$\leq$ 32B	$\leq$ 16B
Pipeline Depth	$<$ 8 Joins	$<$ 2 Joins
Skew (Zipf)	≤ 1	≤ 0.5

Build complex dataflows from simple components

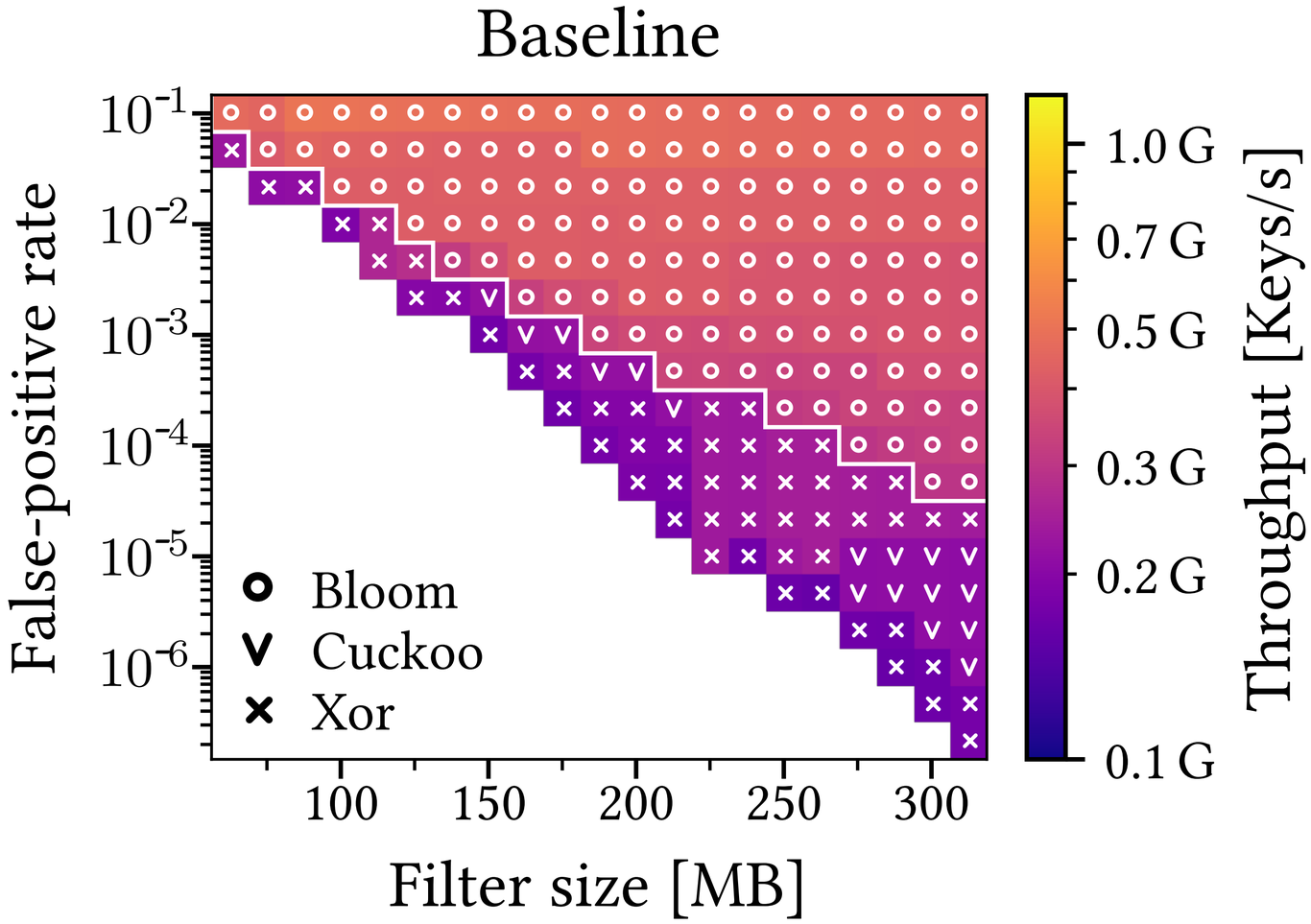
Use simple operators to build a hash join



Hardware-accelerated ***partitioner*** or ***Smart NIC*** between storage and compute layer

Best-Performing Filter

Baseline without partitioning or vectorization



Approximate Filters

with comparable optimizations

- **Bloom Filter Variants**

  Naive, Blocked, (Cache-)Sectorized

- **Fingerprint Filter**

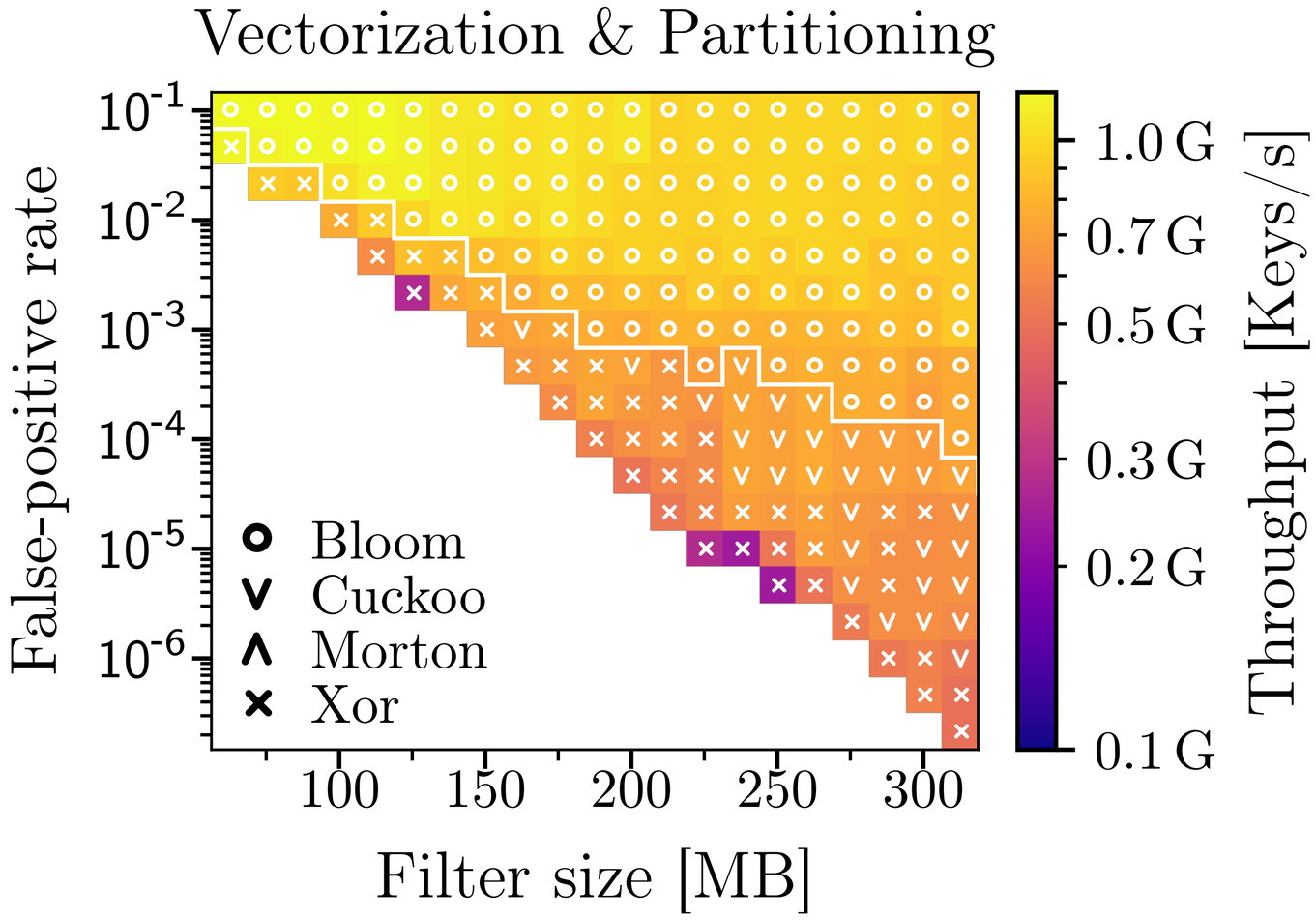
  Cuckoo, Xor, Morton

Dataset

- 100 M random keys
- 64-bit Integers

Impact of the Vectorization & Partitioning

Fingerprint-based filters profit much more



Vectorization:

Bloom +68%

Cuckoo +68%

Xor +17%

Vectorization & Partitioning:

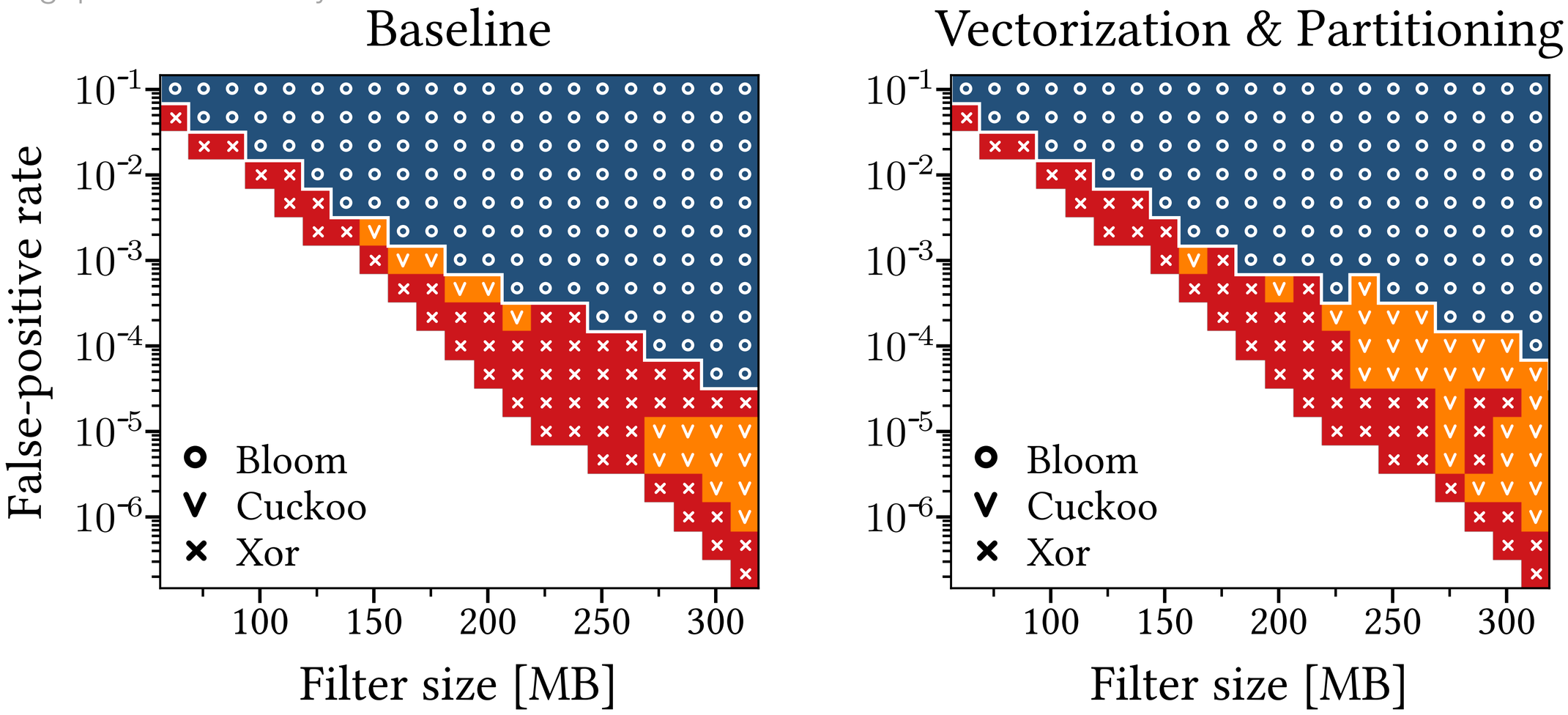
Bloom +130%

Cuckoo +210%

Xor +203%

Best-performing Filter

Throughput vs. Versatility



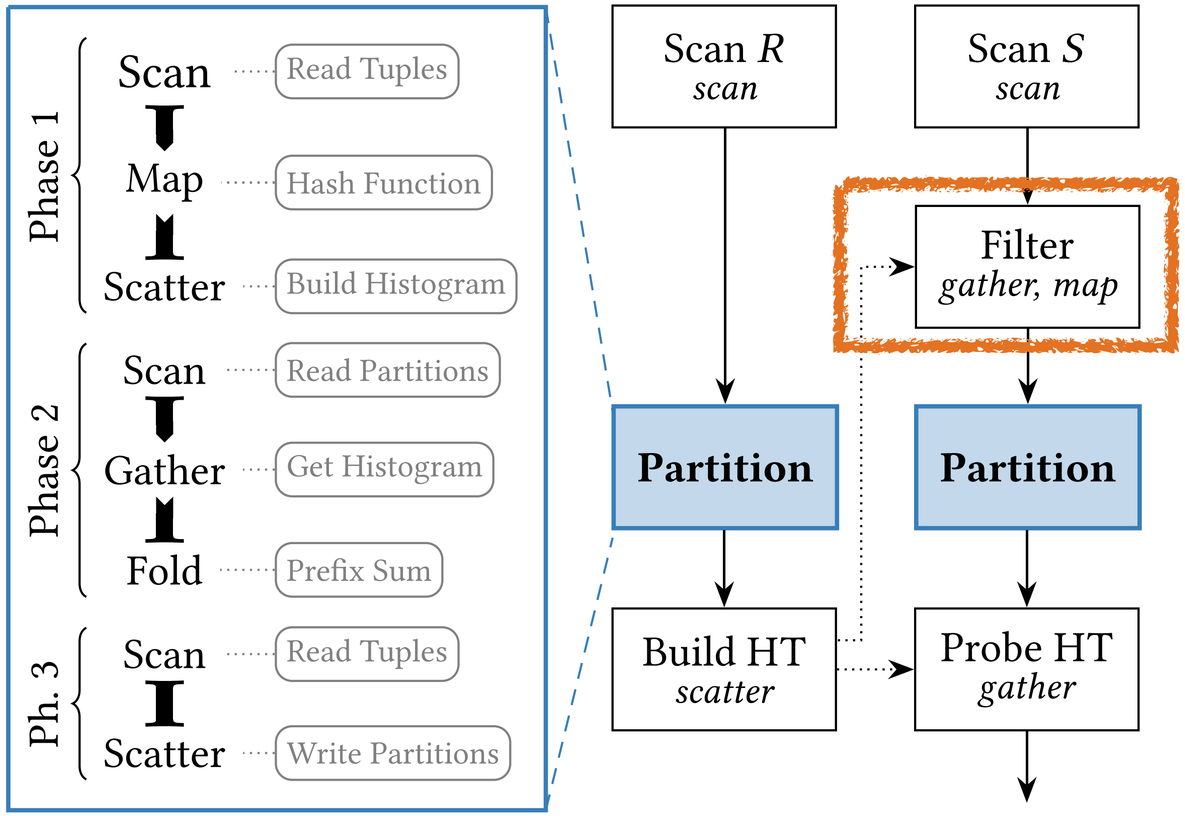
Bloom filter provides highest throughput

⟺

Fingerprint filters are smaller and more versatile

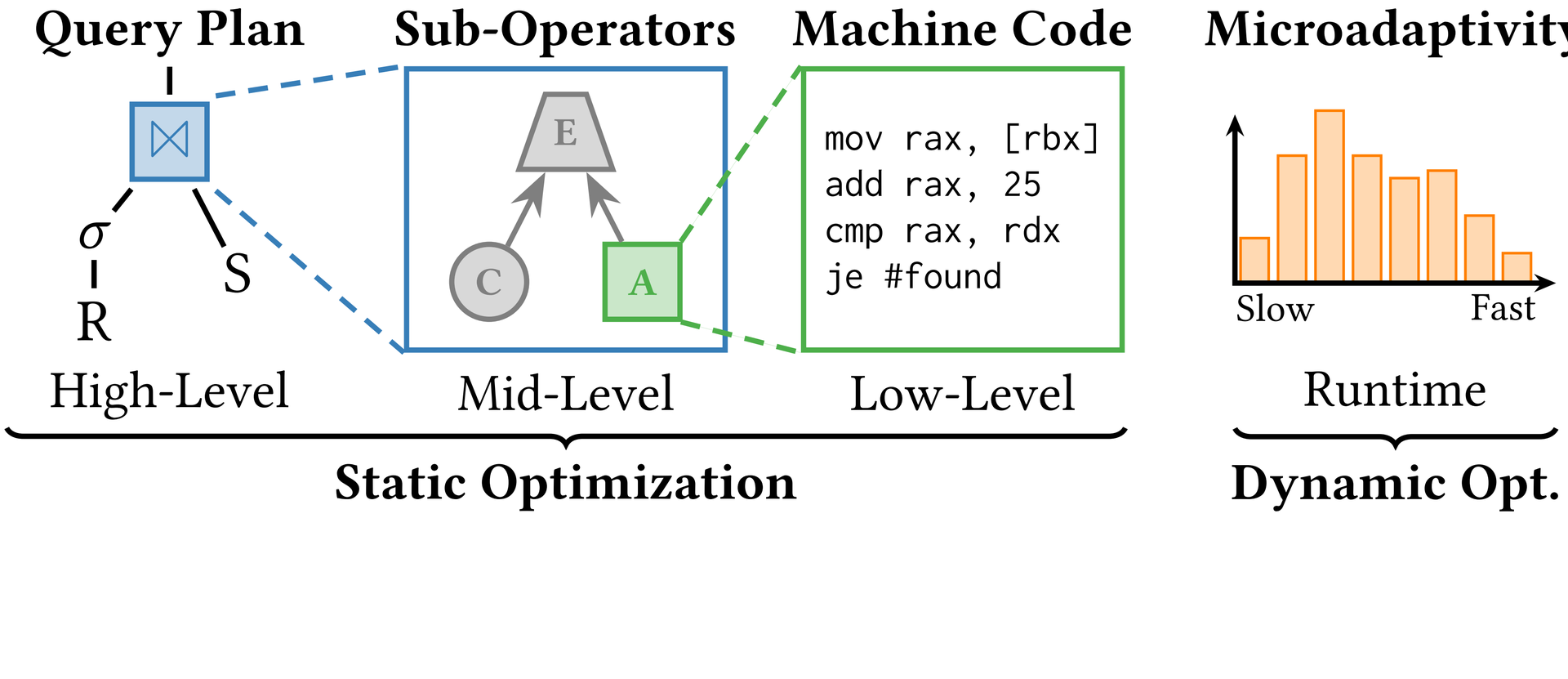
Offload dataflows to different targets and accelerators

Use the fine granularity to offload in chunks



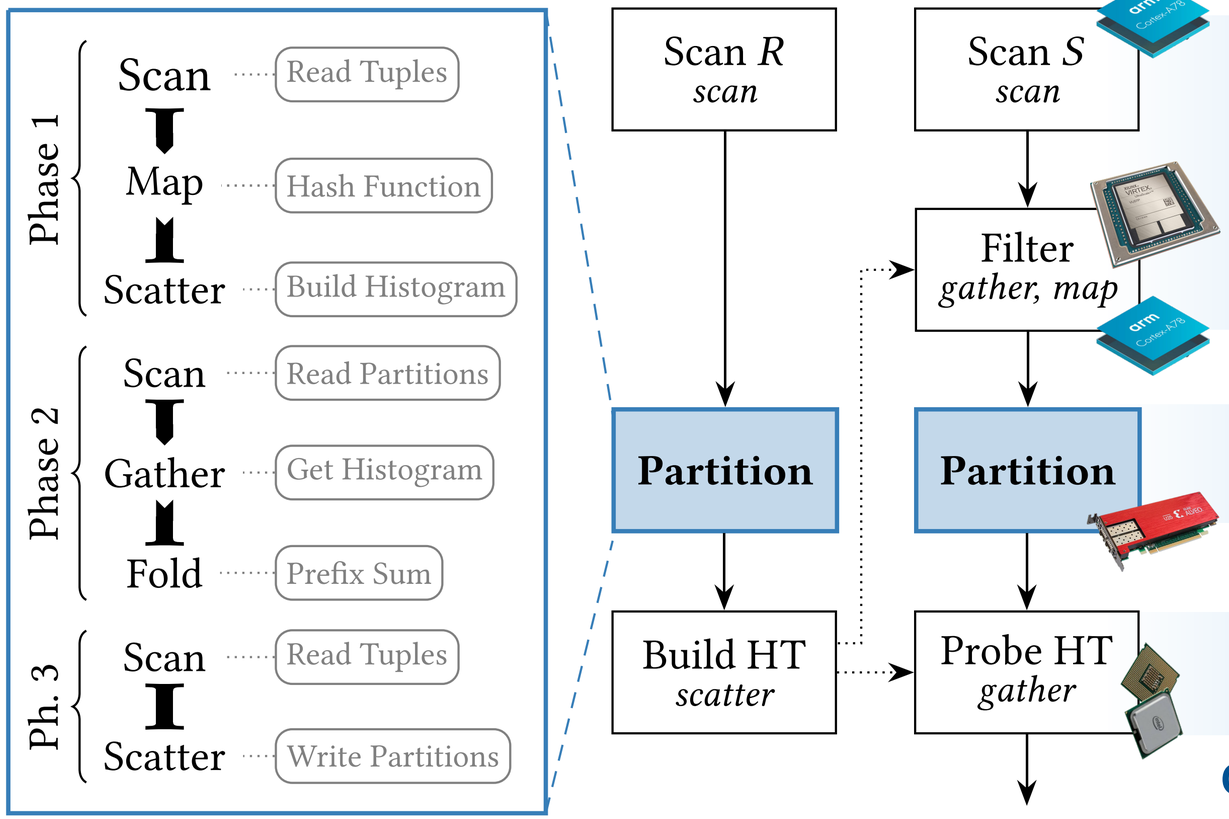
Optimize queries on different layers

Reuse existing query optimizer input and extend to disaggregated resources



Offload dataflows to different targets and accelerators

Use the fine granularity to offload in chunks



Storage layer (e.g., ARM/FPGAs)

Parallel high-bandwidth ***scan*** of relation S,
and
a semi-join reducer (***filter***)
on ***computational storage device***

(Radix-/Range-) ***partition*** the data on the
Smart NIC before reaching the compute layer

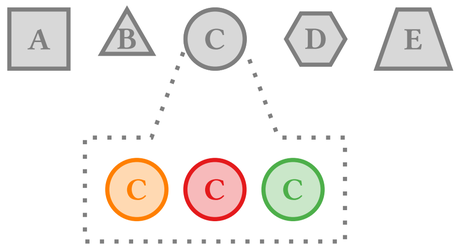
Probe the HT on the **compute** host **layer**

Compute layer (e.g., x86/ARM/GPUs)

Take-Away Messages

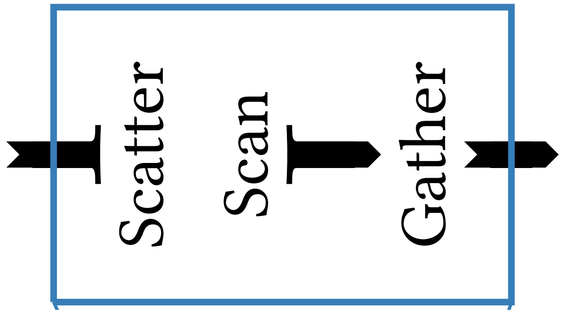
Benefits of sub-operators

Fine-Granular Building Blocks



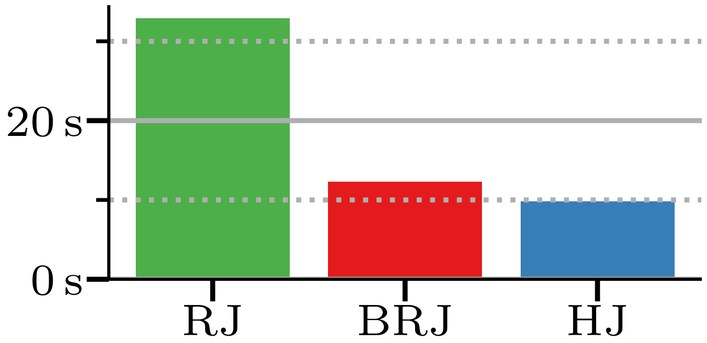
=> Offload to Modern Hardware Targets

Notion of Materialization Points



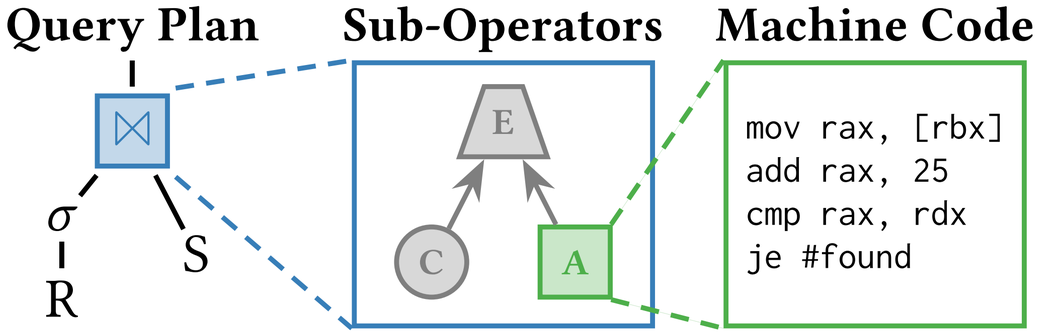
=> Future Proof for Resource Disaggregation

CPU-Partitioning very sensitive to workload



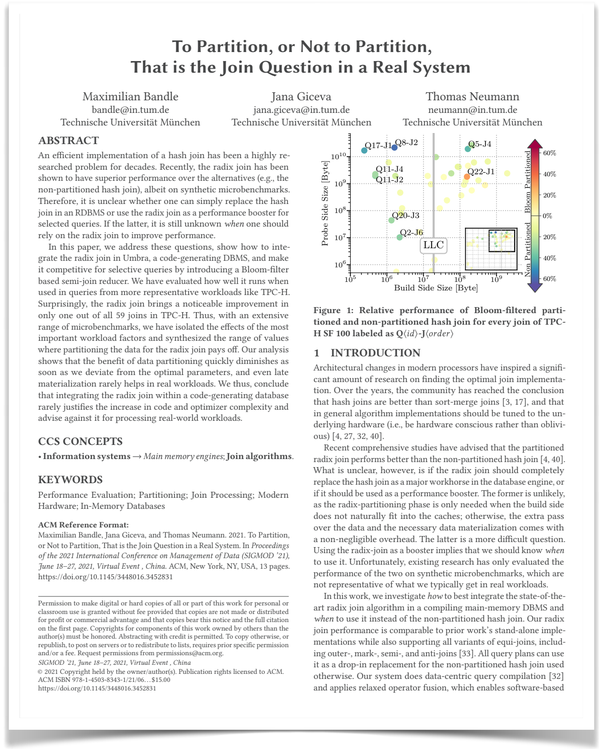
=> Use HW partitioner or trust CPU caches

Filtering alone is already tricky too optimize



=> Multi-Level Optimizer

Read more in the papers: db.in.tum.de/~bandle



To Partition, or Not to Partition,
That is the Join Question in a Real System

Maximilian Bandle
bandle@in.tum.de
Technische Universität München

Jana Giceva
jana.giceva@in.tum.de
Technische Universität München

Thomas Neumann
neumann@in.tum.de
Technische Universität München

ABSTRACT

An efficient implementation of a hash join has been a highly researched problem for decades. Recently, the radix join has been shown to have superior performance over the alternatives (e.g., the non-partitioned hash join), albeit on synthetic microbenchmarks. Therefore, it is unclear whether one can simply replace the hash join in an RDBMS or use the radix join as a performance booster for selected queries. If the latter, it is still unknown *when* one should rely on the radix join to improve performance.

In this paper, we address these questions, show how to integrate the radix join in Umbra, a code-generating DBMS, and make it competitive for selective queries by introducing a Bloom-filter based semi-join reducer. We have evaluated how well it runs when used in queries from more representative workloads like TPC-H. Surprisingly, the radix join brings a noticeable improvement in only one out of all 59 joins in TPC-H. Thus, with an extensive range of microbenchmarks, we have isolated the effects of the most important workload factors and synthesized the range of values where partitioning the data for the radix join pays off. Our analysis shows that the benefit of data partitioning quickly diminishes as soon as we deviate from the optimal parameters, and even late materialization rarely helps in real workloads. We thus, conclude that integrating the radix join within a code-generating database rarely justifies the increase in code and optimizer complexity and advise against it for processing real-world workloads.

CCS CONCEPTS

• **Information systems** → *Main memory engines*; **Join algorithms**.

KEYWORDS

Performance Evaluation; Partitioning; Join Processing; Modern Hardware; In-Memory Databases

ACM Reference Format:
Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event , China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452831

SIGMOD '21, June 18–27, 2021, Virtual Event , China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452831

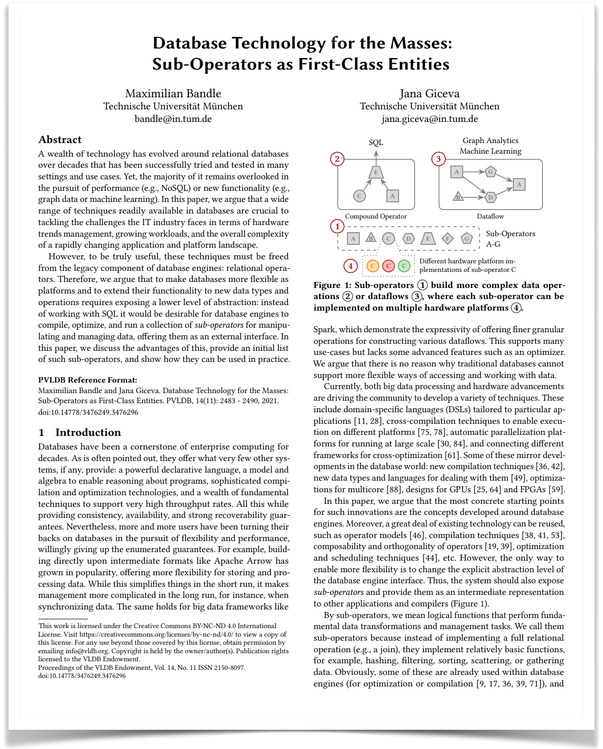
Figure 1: Relative performance of Bloom-filtered partitioned and non-partitioned hash join for every join of TPC-H SF 100 labeled as Q(*id*)-J(*order*)

1 INTRODUCTION

Architectural changes in modern processors have inspired a significant amount of research on finding the optimal join implementation. Over the years, the community has reached the conclusion that hash joins are better than sort-merge joins [3, 17], and that in general algorithm implementations should be tuned to the underlying hardware (i.e., be hardware conscious rather than oblivious) [4, 27, 32, 40].

Recent comprehensive studies have advised that the partitioned radix join performs better than the non-partitioned hash join [4, 40]. What is unclear, however, is if the radix join should completely replace the hash join as a major workhorse in the database engine, or if it should be used as a performance booster. The former is unlikely, as the radix-partitioning phase is only needed when the build side does not naturally fit into the caches; otherwise, the extra pass over the data and the necessary data materialization comes with a non-negligible overhead. The latter is a more difficult question. Using the radix-join as a booster implies that we should know *when* to use it. Unfortunately, existing research has only evaluated the performance of the two on synthetic microbenchmarks, which are not representative of what we typically get in real workloads.

In this work, we investigate *how* to best integrate the state-of-the-art radix join algorithm in a compiling main-memory DBMS and *when* to use it instead of the non-partitioned hash join. Our radix join performance is comparable to prior work's stand-alone implementations while also supporting all variants of equi-joins, including outer-, mark-, semi-, and anti-joins [33]. All query plans can use it as a drop-in replacement for the non-partitioned hash join used otherwise. Our system does data-centric query compilation [32] and applies relaxed operator fusion, which enables software-based



Database Technology for the Masses:
Sub-Operators as First-Class Entities

Maximilian Bandle
Technische Universität München
bandle@in.tum.de

Jana Giceva
Technische Universität München
jana.giceva@in.tum.de

Abstract

A wealth of technology has evolved around relational databases over decades that has been successfully tried and tested in many settings and use cases. Yet, the majority of it remains overlooked in the pursuit of performance (e.g., NoSQL) or new functionality (e.g., graph data or machine learning). In this paper, we argue that a wide range of techniques readily available in databases are crucial to tackling the challenges the IT industry faces in terms of hardware trends management, growing workloads, and the overall complexity of a rapidly changing application and platform landscape.

However, to be truly useful, these techniques must be freed from the legacy component of database engines: relational operators. Therefore, we argue that to make databases more flexible as platforms and to extend their functionality to new data types and operations requires exposing a lower level of abstraction: instead of working with SQL it would be desirable for database engines to compile, optimize, and run a collection of *sub-operators* for manipulating and managing data, offering them as an external interface. In this paper, we discuss the advantages of this, provide an initial list of such sub-operators, and show how they can be used in practice.

PVLDB Reference Format:
Maximilian Bandle and Jana Giceva. Database Technology for the Masses: Sub-Operators as First-Class Entities. PVLDB, 14(11): 2483 - 2490, 2021.
doi:10.14778/3476249.3476296

1 Introduction

Databases have been a cornerstone of enterprise computing for decades. As is often pointed out, they offer what very few other systems, if any, provide: a powerful declarative language, a model and algebra to enable reasoning about programs, sophisticated compilation and optimization technologies, and a wealth of fundamental techniques to support very high throughput rates. All this while providing consistency, availability, and strong recoverability guarantees. Nevertheless, more and more users have been turning their backs on databases in the pursuit of flexibility and performance, willingly giving up the enumerated guarantees. For example, building directly upon intermediate formats like Apache Arrow has grown in popularity, offering more flexibility for storing and processing data. While this simplifies things in the short run, it makes management more complicated in the long run, for instance, when synchronizing data. The same holds for big data frameworks like

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476296

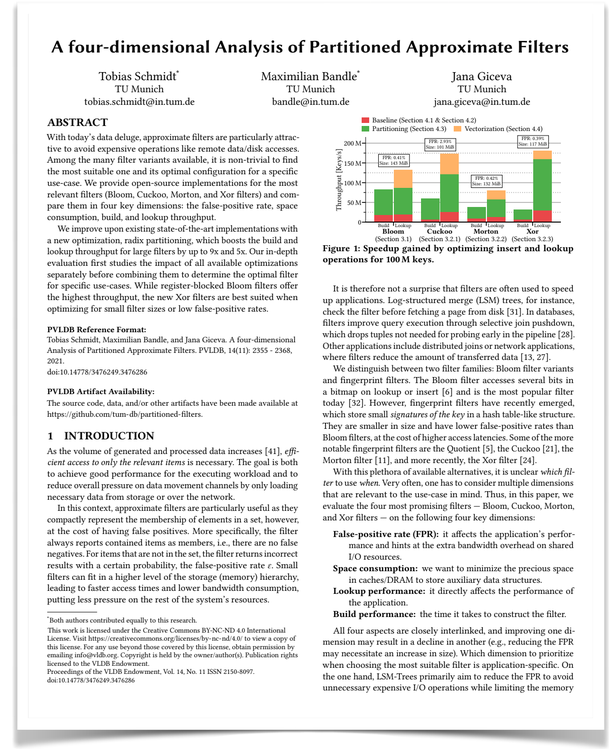
Figure 1: Sub-operators ① build more complex data operations ② or dataflows ③, where each sub-operator can be implemented on multiple hardware platforms ④.

Spark, which demonstrate the expressivity of offering finer granular operations for constructing various dataflows. This supports many use-cases but lacks some advanced features such as an optimizer. We argue that there is no reason why traditional databases cannot support more flexible ways of accessing and working with data.

Currently, both big data processing and hardware advancements are driving the community to develop a variety of techniques. These include domain-specific languages (DSLs) tailored to particular applications [11, 28], cross-compilation techniques to enable execution on different platforms [75, 78], automatic parallelization platforms for running at large scale [30, 84], and connecting different frameworks for cross-optimization [61]. Some of these mirror developments in the database world: new compilation techniques [36, 42], new data types and languages for dealing with them [49], optimizations for multicore [88], designs for GPUs [25, 64] and FPGAs [59].

In this paper, we argue that the most concrete starting points for such innovations are the concepts developed around database engines. Moreover, a great deal of existing technology can be reused, such as operator models [46], compilation techniques [38, 41, 53], composability and orthogonality of operators [19, 39], optimization and scheduling techniques [44], etc. However, the only way to enable more flexibility is to change the explicit abstraction level of the database engine interface. Thus, the system should also expose *sub-operators* and provide them as an intermediate representation to other applications and compilers (Figure 1).

By sub-operators, we mean logical functions that perform fundamental data transformations and management tasks. We call them sub-operators because instead of implementing a full relational operation (e.g., a join), they implement relatively basic functions, for example, hashing, filtering, sorting, scattering, or gathering data. Obviously, some of these are already used within database engines (for optimization or compilation [9, 17, 36, 39, 71]), and



A four-dimensional Analysis of Partitioned Approximate Filters

Tobias Schmidt*
TU Munich
tobias.schmidt@in.tum.de

Maximilian Bandle*
TU Munich
bandle@in.tum.de

Jana Giceva
TU Munich
jana.giceva@in.tum.de

ABSTRACT

With today's data deluge, approximate filters are particularly attractive to avoid expensive operations like remote data/disk accesses. Among the many filter variants available, it is non-trivial to find the most suitable one and its optimal configuration for a specific use-case. We provide open-source implementations for the most relevant filters (Bloom, Cuckoo, Morton, and Xor filters) and compare them in four key dimensions: the false-positive rate, space consumption, build, and lookup throughput.

We improve upon existing state-of-the-art implementations with a new optimization, radix partitioning, which boosts the build and lookup throughput for large filters by up to 9x and 5x. Our in-depth evaluation first studies the impact of all available optimizations separately before combining them to determine the optimal filter for specific use-cases. While register-blocked Bloom filters offer the highest throughput, the new Xor filters are best suited when optimizing for small filter sizes or low false-positive rates.

PVLDB Reference Format:
Tobias Schmidt, Maximilian Bandle, and Jana Giceva. A four-dimensional Analysis of Partitioned Approximate Filters. PVLDB, 14(11): 2355 - 2368, 2021.
doi:10.14778/3476249.3476286

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tum-db/partitioned-filters.

1 INTRODUCTION

As the volume of generated and processed data increases [41], *efficient access to only the relevant items* is necessary. The goal is both to achieve good performance for the executing workload and to reduce overall pressure on data movement channels by only loading necessary data from storage or over the network.

In this context, approximate filters are particularly useful as they compactly represent the membership of elements in a set, however, at the cost of having false positives. More specifically, the filter always reports contained items as members, i.e., there are no false negatives. For items that are not in the set, the filter returns incorrect results with a certain probability, the false-positive rate ε. Small filters can fit in a higher level of the storage (memory) hierarchy, leading to faster access times and lower bandwidth consumption, putting less pressure on the rest of the system's resources.

Figure 1: Speedup gained by optimizing insert and lookup operations for 100 M keys.

It is therefore not a surprise that filters are often used to speed up applications. Log-structured merge (LSM) trees, for instance, check the filter before fetching a page from disk [31]. In databases, filters improve query execution through selective join pushdown, which drops tuples not needed for probing early in the pipeline [28]. Other applications include distributed joins or network applications, where filters reduce the amount of transferred data [13, 27].

We distinguish between two filter families: Bloom filter variants and fingerprint filters. The Bloom filter accesses several bits in a bitmap on lookup or insert [6] and is the most popular filter today [32]. However, fingerprint filters have recently emerged, which store small *signatures of the key* in a hash table-like structure. They are smaller in size and have lower false-positive rates than Bloom filters, at the cost of higher access latencies. Some of the more notable fingerprint filters are the Quotient [5], the Cuckoo [21], the Morton filter [11], and more recently, the Xor filter [24].

With this plethora of available alternatives, it is unclear *which filter* to use *when*. Very often, one has to consider multiple dimensions that are relevant to the use-case in mind. Thus, in this paper, we evaluate the four most promising filters — Bloom, Cuckoo, Morton, and Xor filters — on the following four key dimensions:

- **False-positive rate (FPR):** it affects the application's performance and hints at the extra bandwidth overhead on shared I/O resources.
- **Space consumption:** we want to minimize the precious space in caches/DRAM to store auxiliary data structures.
- **Lookup performance:** it directly affects the performance of the application.
- **Build performance:** the time it takes to construct the filter.

All four aspects are closely interlinked, and improving one dimension may result in a decline in another (e.g., reducing the FPR may necessitate an increase in size). Which dimension to prioritize when choosing the most suitable filter is application-specific. On the one hand, LSM-Trees primarily aim to reduce the FPR to avoid unnecessary expensive I/O operations while limiting the memory

* Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476286

To partition, or not to partition, that is the join question in a real system

Bandle, Giceva & Neumann
SIGMOD '21

Database Technology for the Masses: Sub-Operators as First-Class Entities

Bandle & Giceva
VLDB '21

A four-dimensional Analysis of Partitioned Approximate Filters

Schmidt*, Bandle* & Giceva
VLDB '21