CS 764: Topics in Database Management Systems

Lecture 6: Buffer Management

Xiangyao Yu

9/23/2025

Today's Paper: Buffer Management

Robust External Hash Aggregation in the Solid State Age

Laurens Kuiper
CWI, Amsterdam, Netherlands
laurens.kuiper@cwi.nl

Peter Boncz
CWI, Amsterdam, Netherlands
peter.boncz@cwi.nl

Hannes Mühleisen
CWI, Amsterdam, Netherlands
hannes.muehleisen@cwi.nl

***Abstract*—Analytical database systems offer high-performance in-memory aggregation. If there are many unique groups, temporary query intermediates may not fit RAM, requiring the use of external storage. However, switching from an in-memory to an external algorithm can degrade performance sharply.**

We revisit external hash aggregation on modern hardware, aiming instead for robust performance that avoids a "performance cliff" when memory runs out.

To achieve this, we introduce two techniques for handling temporary query intermediates. First, we propose unifying the memory management of temporary and persistent data. Second, we propose using a page layout that can be spilled to disk despite being optimized for main memory performance. These two techniques allow operator implementations to process larger-than-memory query intermediates with only minor modifications.

We integrate these into DuckDB's parallel hash aggregation. Experimental results show that our implementation gracefully degrades performance as query intermediates exceed the available memory limit, while main memory performance is competitive with other analytical database systems.

***Index Terms*—relational databases, database query processing, aggregation**

I. Introduction

Until late in the 20th century, main memory was expensive; therefore, traditional database management systems (DBMS) optimized for disk access, as this was their major bottleneck. "Spillable" data structures like B-trees [1] were used not only to speed up retrieval of persistent data but also inside query operators. As a result, these systems could process workloads that were larger than the small amount of available memory.

Around the 2000s, RAM prices decreased, and database systems optimized for main memory [2], for both persistent data and temporary query intermediates [3]. In these systems, main memory access became the bottleneck, and techniques were devised to make better use of CPU caches [4]. DBMSes have now evolved into large monolithic database servers, often with large amounts of RAM at their disposal.

Pure in-memory systems are not economical, however. Efficient utilization of secondary storage, e.g., by caching, is key to providing good performance at a low cost [5]. In recent years, there has been a renewed interest in buffer management, specifically for solid-state memory, that offers much higher bandwidth and lower latency than magnetic disk [6]–[8]. Data management systems are now reverting to being disk-based without sacrificing in-memory performance [9].

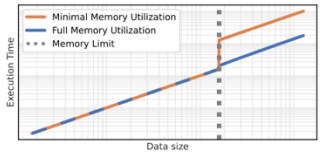


Fig. 1. Conceptual aggregation performance vs data size (log-log scale). When switching from an in-memory strategy to an external strategy that minimizes memory usage, performance degradation is harsh and sudden (a "performance cliff"). A unified strategy for in-memory and external aggregation that utilizes all available memory degrades more gracefully (performance-robust).

This body of research has focused on using storage for persistent data but, for the most part, ignored temporary query intermediates. Analytical (OLAP) systems, which frequently process large volumes of data and often have large query intermediates, became mainstream after DBMSes optimized for main memory. As systems became able to process queries on arbitrary-sized persistent tables, intermediate results can - depending on the query - also grow to arbitrary sizes. In these cases, many modern OLAP systems either abort queries or switch to a traditional disk-based algorithm that is orders of magnitude slower, introducing a "performance cliff", as illustrated in Figure 1.

Given the advancements of OLAP systems in the past two decades [10]–[12], and the research into buffer management on modern hardware [6], [8], OLAP systems should be able to perform more robustly when intermediates exceed main memory. However, traditional buffer managers have fixed-size pages and a statically allocated pool. This inflexibility makes them undesirable for intermediates. Therefore, temporary data is allocated differently. Managing the entire memory pool, i.e., persistent and temporary data, in a cooperative manner may help systems better utilize available memory [13].

In this paper, we go beyond Cooperative Memory Management and take a *unified* approach to memory management for persistent and temporary data. We have developed a specialized page layout specifically for temporary data to accommodate this. We have integrated this into the hash aggregation operator of DuckDB, our in-process analytical

ICDE 2024

Algorithmica (1986) 1: 311–336

Algorithmica

An Evaluation of Buffer Management Strategies for Relational Database Systems[1]

Hong-Tai Chou[2,3] and David J. DeWitt[2]

Abstract. In this paper we present a new algorithm, DBMIN, for managing the buffer pool of a relational database management system. DBMIN is based on a new model of relational query behavior, the *query locality set model* (QLSM). Like the hot set model, the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm. After introducing the QLSM and describing the DBMIN algorithm, we present a performance evaluation methodology for evaluating buffer management algorithms in a multiuser environment. This methodology employed a hybrid model that combines features of both trace-driven and distribution-driven simulation models. Using this model, the performance of the DBMIN algorithm in a multiuser environment is compared with that of the hot set algorithm and four more traditional buffer replacement algorithms.

Key Words. Buffer management, Database systems, Page replacement strategies, Hybrid simulation, Performance evaluation.

1. Introduction. In this paper we present a new algorithm, DBMIN, for managing the buffer pool of a relational database management system. DBMIN is based on a new model of relational query behavior, the *query locality set model* (QLSM.) Like the hot set model [Sacc 1], the QLSM has an advantage over stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm. After introducing the QLSM and describing the DBMIN algorithm, the performance of the DBMIN algorithm in a multiuser environment is compared with that of the hot set algorithm and four more traditional buffer replacement algorithms.

A number of factors motivated this research. First, although Stonebraker [Ston 2] convincingly argued that conventional virtual memory page replacement algorithms (e.g., *least recently used* (LRU)) were generally not suitable for a

[1] This research was partially supported by the Department of Energy under Contract No. DE-AC02-81ER10920 and the National Science Foundation under grant MCS82-01870.
[2] Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, USA.
[3] Current Address: Microelectronics and Computer Technology Corporation, Austin, Texas, USA.

Received March 15, 1986; revised July 7, 1986. Communicated by Dale Skeen.

Algorithmica 1986

Outline

Table data vs. temporary data

Buffer management for table data

- Page-based buffer management

Buffer management for temporary data

Unified memory management

Future work

Table Data vs. Temporary Data

Table data

- Data from input tables
- Page-based buffer management is widely used for table data
- Transparently handle data movement between disk and memory

Temporary data

- E.g., hash tables (for join or aggregation), sort buffers, etc.
- Traditionally malloc-based, not in page granularity
- Must explicitly move data between disk and memory

Outline

Table data vs. temporary data

Buffer management for table data

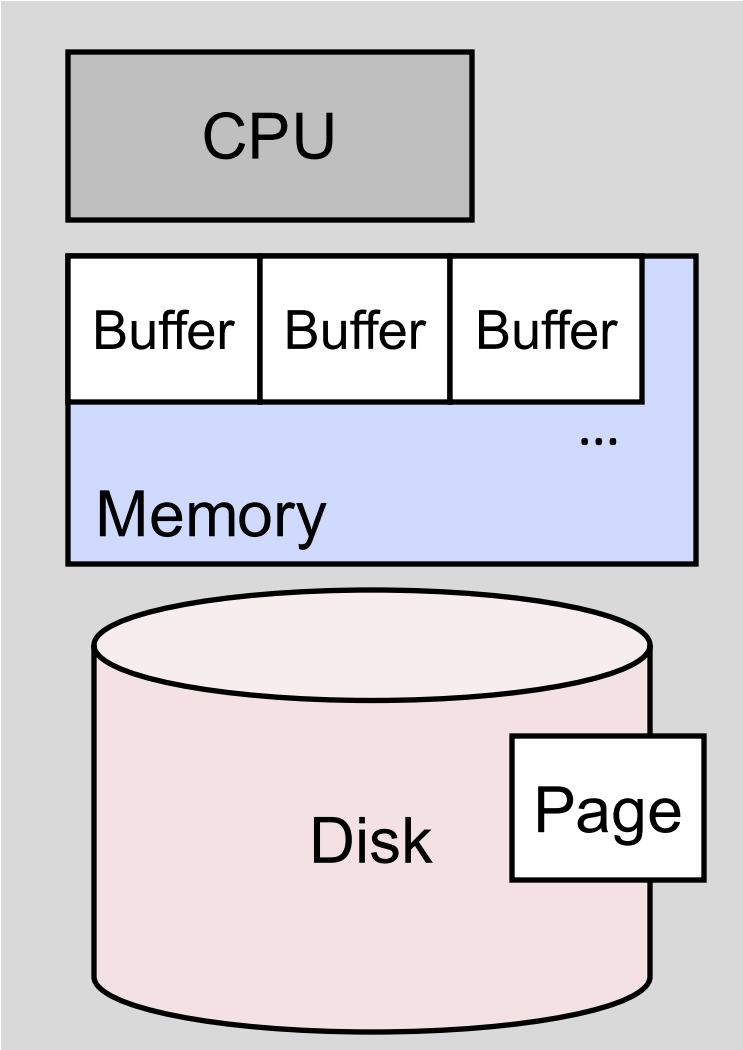
- Page-based buffer management

Buffer management for temporary data

Unified memory management

Future work

System Architecture

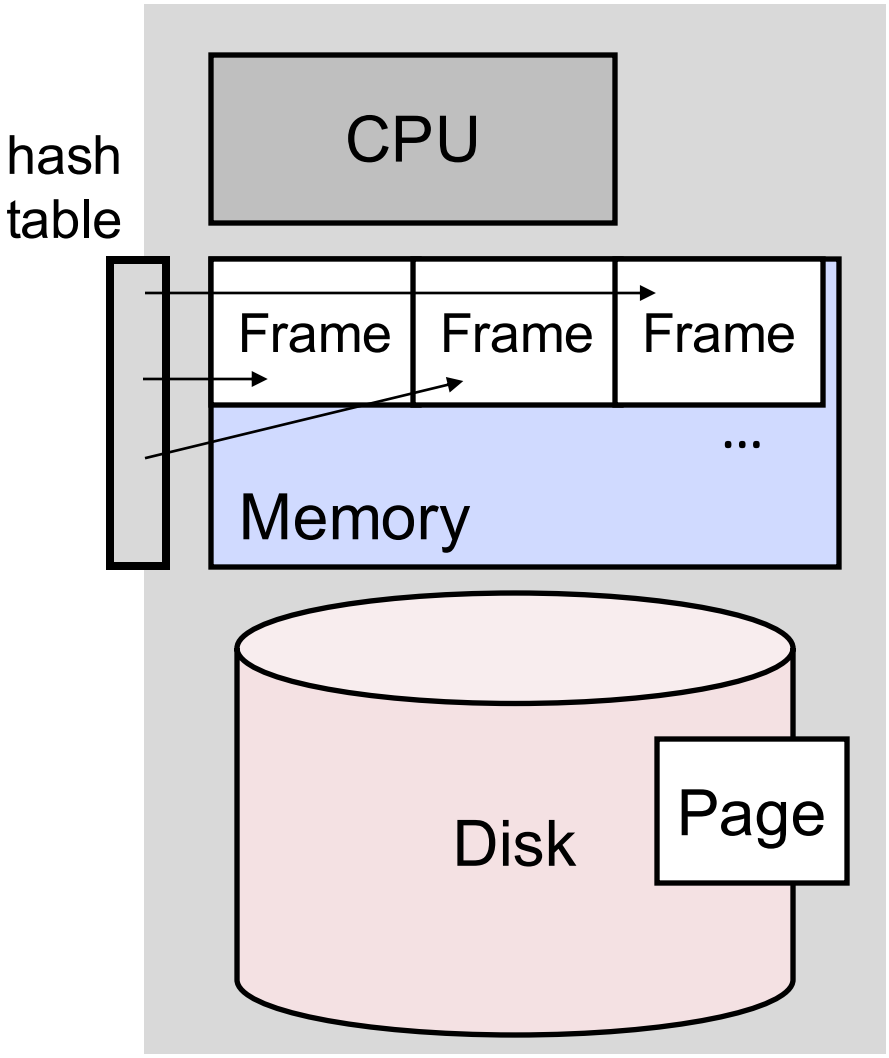


A database management system (DBMS) manipulate data in memory

- Data on disk must be loaded to memory before processed

The unit of data movement is a **page**

Page-Based Buffer Management

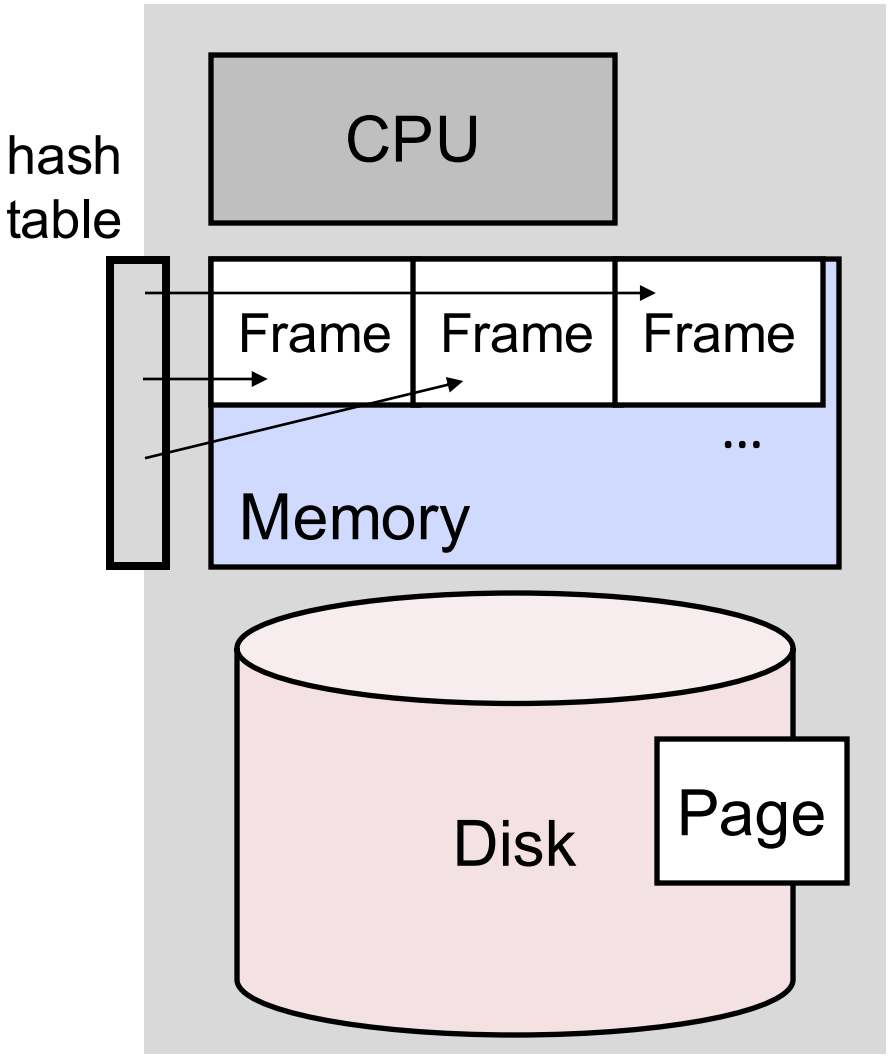


Page granularity: Data managed in page granularity

Indirection: Each page is identified with a **page ID**; a hash table stores whether the page is in memory or on disk.

Page placement is handled by the buffer manager and not controlled by operators

Page-Based Buffer Management



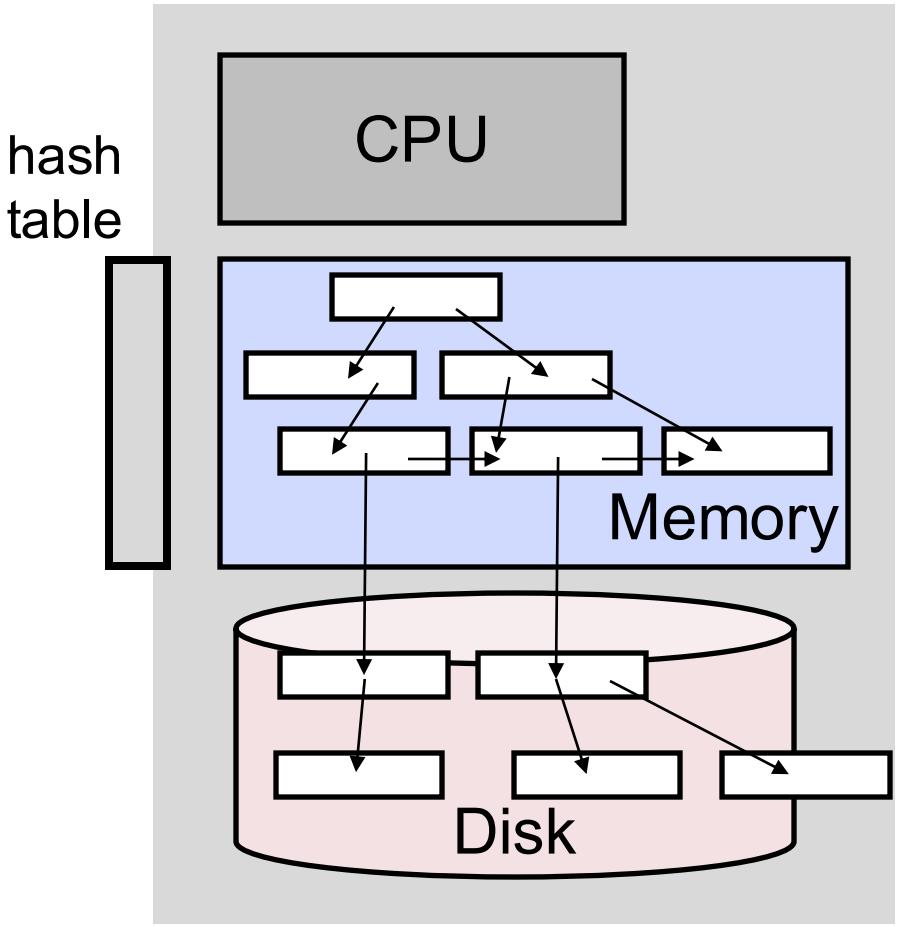


Table data can be organized in a B-tree

Page Replacement Policy

Important question: what pages should stay in memory?

- LRU (Lease recently used)
- Clock
- MRU (Most recently used)
- FIFO, Random, …

Insight: the optimal buffer replacement and allocation policies depend on the data access pattern, which is relatively easy to predict in a DBMS compared to hardware or OS

LRU Replacement

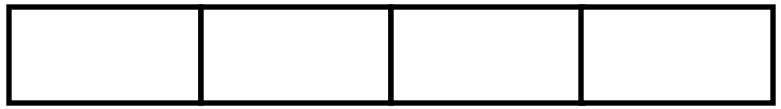
Replace the **least-recently used** (LRU) item in the buffer

Intuition: more recently used items will more likely to be used again in the future

LRU Replacement Example

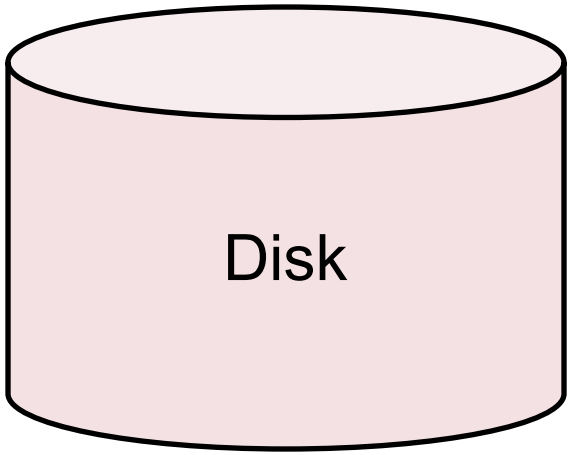
Example: memory contains 4 buffers. LRU replacement policy

Memory



Incoming requests

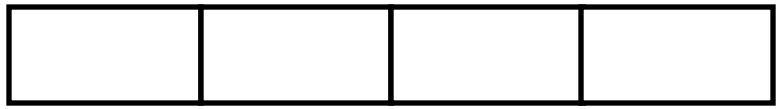
0, 1, 2, 3, 0, 1, 2, 4, 0, 1, 2, 5, …



A Different Access Pattern

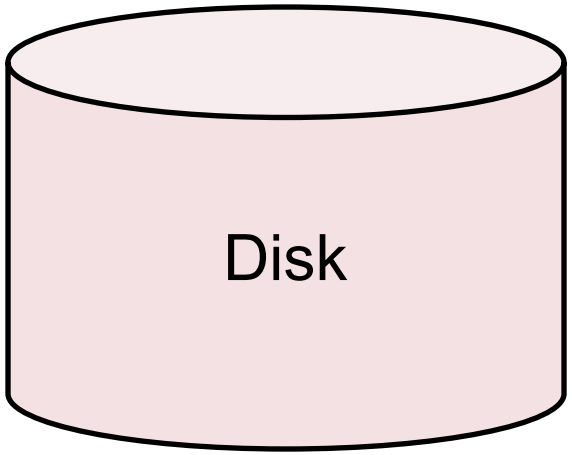
Example: memory contains 4 buffers. LRU replacement policy

Memory



Incoming requests

0, 1, 2, 3, 4, 0, 1, 2, 3, 4, …



MRU Replacement

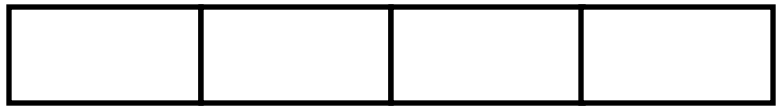
Replace the **most-recently used** (LRU) item in the buffer

Intuition: avoid the cache thrashing problem in the previous example

MRU Replacement Example

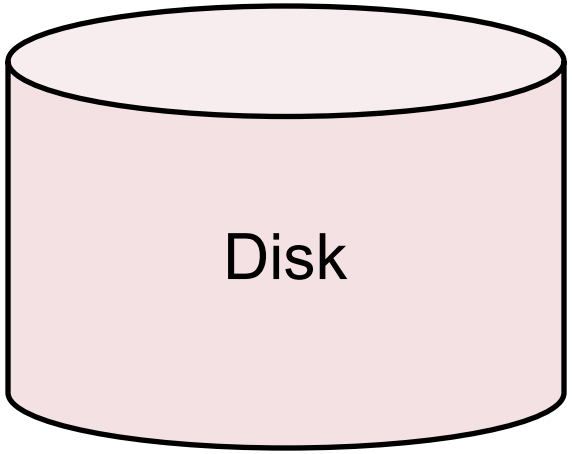
Example: memory contains 4 buffers. MRU replacement policy

Memory



Incoming requests

0, 1, 2, 3, 4, 0, 1, 2, 3, 4, …



Query Locality Set Model

Observations

- DBMS supports a limited set of operations
- Data reference patterns are regular and predictable
- Complex reference patterns can be decomposed into simple patterns

Buffer allocation decisions:

1. **Locality set**: The appropriate buffer pool size for a query
2. **Replacement policy**

QLSM – Sequential References

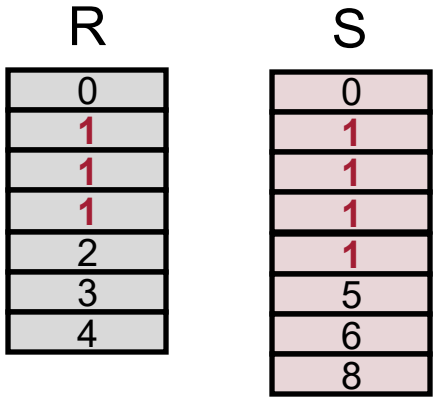
Straight sequential (SS): each page in a file accessed only once

- E.g., select on an unordered relation
- Locality set: one page
- Replacement policy: any

QLSM – Sequential References

Straight sequential (SS): each page in a file accessed only once
- E.g., select on an unordered relation
- Locality set: one page
- Replacement policy: any

Clustered sequential (CS): repeatedly read a “chunk” sequentially
- E.g., sort-merge join with duplicate join keys
- Locality set: size of largest cluster
- Replacement policy: LRU or FIFO (buffer size ≥ cluster size), MRU (otherwise)



QLSM – Sequential References

Straight sequential (SS): each page in a file accessed only once
- E.g., select on an unordered relation
- Locality set: one page
- Replacement policy: any

Clustered sequential (CS): repeatedly read a “chunk” sequentially
- E.g., sort-merge join with duplicate join keys
- Locality set: size of largest cluster
- Replacement policy: LRU or FIFO (buffer size ≥ cluster size), MRU (otherwise)

Looping Sequential (LS): repeatedly read something sequentially
- E.g. nested-loop join
- Locality set: size of the file being repeated scanned.
- Replacement policy: MRU

QLSM – Random References

Independent random (IR): truly random accesses

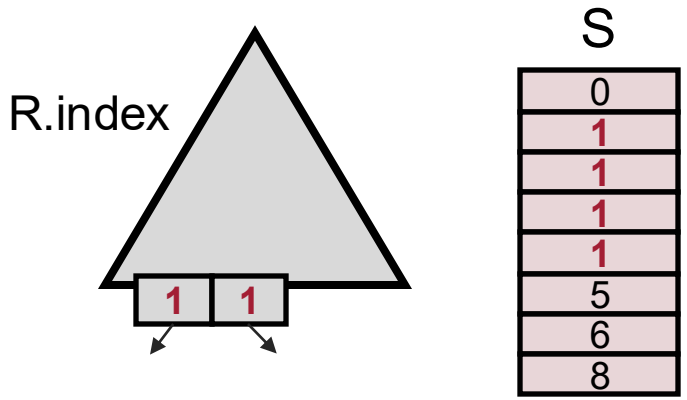
- E.g., index scan through a non-clustered (e.g., secondary) index
- Locality set: one page or **b** pages (**b** unique pages are accessed in total)
- Replacement: any

Clustered random (CR): random accesses with some locality

- E.g., join between non-clustered, non-unique index as inner relation and clustered, non-unique outer relation
- Locality set: size of the largest cluster
- Replacement policy :

  LRU or FIFO (buffer size ≥ cluster size)

  MRU (otherwise)



Outline

Table data vs. temporary data

Buffer management for table data

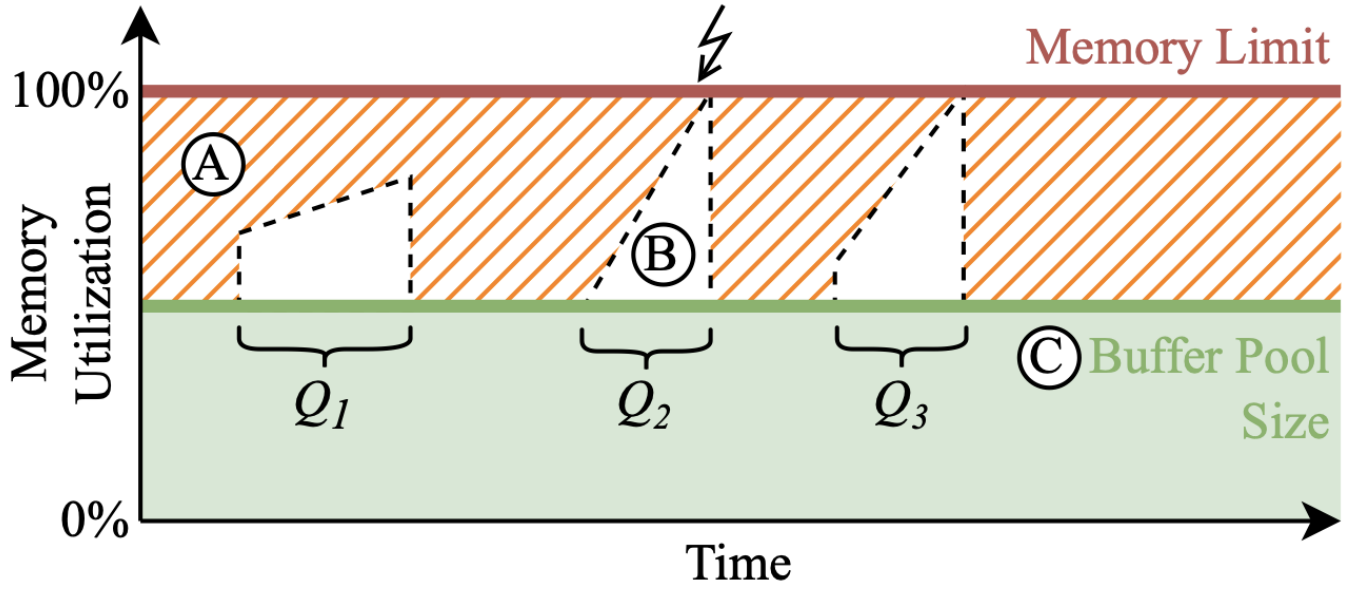
- Page-based buffer management

Buffer management for temporary data

Unified memory management

Future work

Traditional DB: Two Memory Pools



Sizes of pools are **statically determined**

Page-based buffer pool for persistent data

- Data spilling is implicitly handled by buffer manager

Malloc-based buffer pool for temporary data

- Data spilling is not supported, or
- Data spilling is explicitly handled by operator

Outline

Table data vs. temporary data

Buffer management for table data

- Page-based buffer management

Buffer management for temporary data

Unified memory management

Future work

DuckDB Unified Memory Management

Insight: Try to use page-based data structures as much as possible

- Operators (e.g., group-by aggregation) are designed accordingly

Benefits:

- Disk spilling is implicitly managed through buffer manager
- Graceful performance degradation when data exceeds memory

Challenges:

- Allow arbitrary eviction without corrupting pointers
- Some data structures are non-trivial to store in pages (e.g., hash tables)

Allocation Types

Persistent data

- Fixed page size (256 KB)

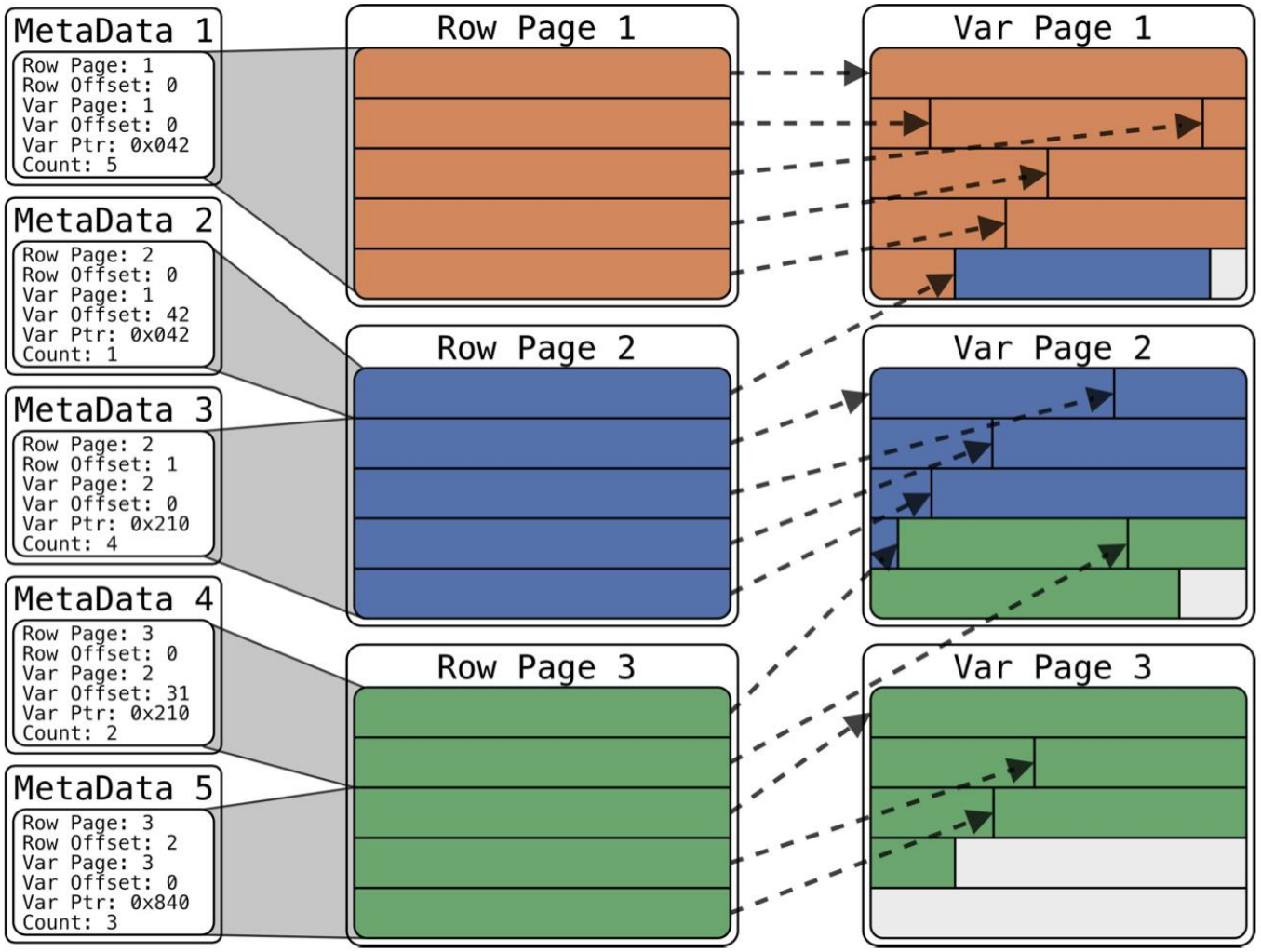
Temporary data

- Non-paged allocations (cannot be spilled)
- **Paged fixed-size allocations => the most common use case**
- Paged variable-size allocations

Page Layout for Variable-Size Row

Design requirements:

- 1) Use a row-major data representation with fixed-size rows
- 2) Store variable-size data on separate pages
- **3) Use explicit addressing for variable-size data**
- 4) Be spillable to storage without additional serialization.



External Hash Aggregation

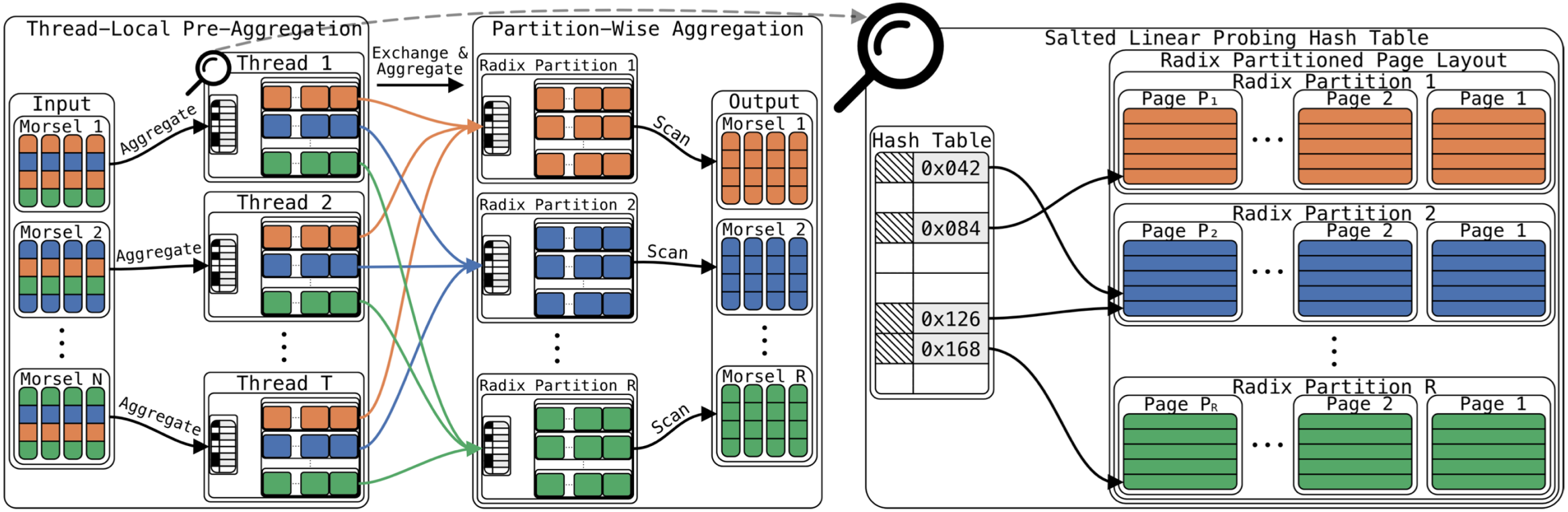


Fig. 3. DuckDB's hash aggregation. Morsels are assigned to threads until all input data has been read. During phase one, each thread pre-aggregates data in a small fixed-size linear probing hash table with one level of indirection, i.e., offsets obtained from hashes access an array of pointers pointing to tuples. Tuples are radix partitioned and stored using DuckDB's spillable page layout, enabling larger-than-memory aggregation. After pre-aggregation, partitions are exchanged and aggregated partition-wise in parallel. Fully aggregated partitions are immediately scanned, effectively becoming morsels in the next pipeline.

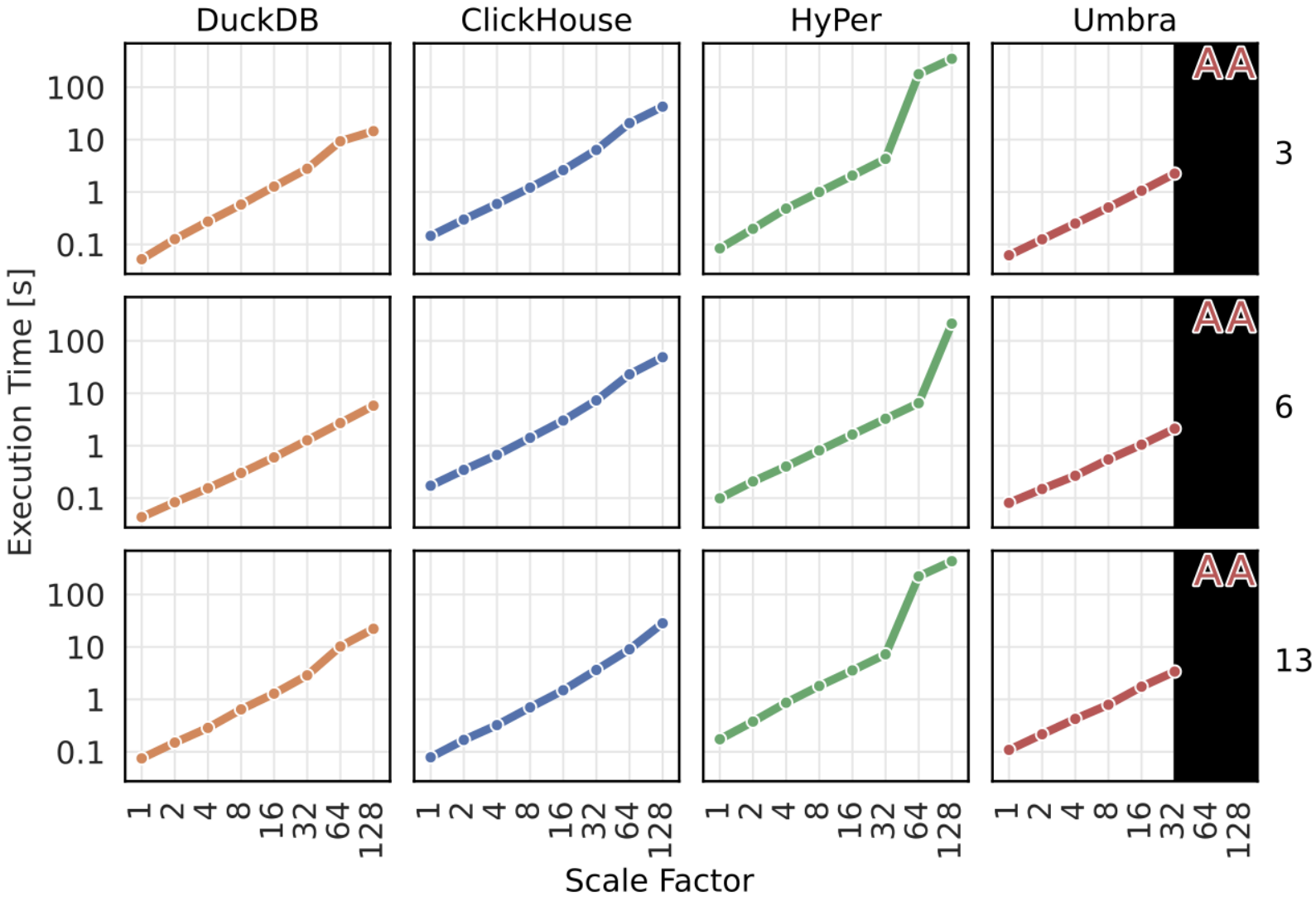
Evaluation – Thin Variant

TABLE II

EXECUTION TIME IN SECONDS FOR THE THIN VARIANT OF ALL GROUPINGS AT SCALE FACTORS 2, 8, 32, AND 128. LOWER IS BETTER. THE LOWEST EXECUTION TIMES ARE HIGHLIGHTED IN BOLD. 'A' DENOTES THAT THE QUERY WAS ABORTED. WE SUMMARIZE BY NORMALIZING EXECUTION TIMES TO DUCKDB AND THEN TAKING THE GEOMETRIC MEAN.

SF	2				8				32				128			
System	Du	Cl	Hy	Um	Du	Cl	Hy	Um	Du	Cl	Hy	Um	Du	Cl	Hy	Um
Grouping 1	**0.01**	0.08	**0.01**	0.02	**0.03**	0.27	0.05	0.04	**0.14**	1.10	**0.14**	0.16	**0.54**	4.06	**0.54**	A
2	0.08	**0.04**	0.13	**0.04**	0.44	**0.16**	0.66	0.19	2.03	0.75	2.86	**0.68**	10.80	**4.04**	12.83	A
3	**0.13**	0.30	0.20	**0.13**	0.58	1.21	1.00	**0.51**	2.78	6.35	4.28	**2.24**	**14.49**	42.47	348.10	A
4	0.12	0.08	0.16	**0.07**	0.58	0.29	0.83	**0.28**	2.86	1.63	3.38	**1.22**	22.06	**9.80**	231.86	A
5	**0.05**	0.08	0.07	**0.05**	0.18	0.29	0.36	**0.16**	0.74	1.57	1.66	**0.54**	**3.17**	9.41	6.86	A
6	**0.08**	0.35	0.21	0.15	**0.30**	1.42	0.81	0.55	**1.27**	7.32	3.27	2.12	**5.80**	48.79	213.41	A
7	**0.17**	0.37	0.28	0.23	**0.72**	1.56	1.36	0.87	**3.42**	8.32	5.61	4.32	**24.62**	51.57	457.52	A
8	0.23	0.36	0.27	**0.20**	0.97	1.51	1.39	**0.78**	5.25	8.11	5.72	**3.44**	65.97	**49.49**	412.86	A
9	**0.16**	0.37	0.30	0.21	**0.74**	1.53	1.58	0.78	3.59	8.01	6.42	**3.47**	**32.77**	46.51	444.50	A
10	**0.25**	0.50	0.41	0.38	**1.10**	2.06	1.96	1.50	**5.78**	11.34	169.63	14.98	89.27	**77.27**	576.68	A
11	**0.13**	**0.13**	0.37	0.18	0.59	**0.51**	1.71	0.65	2.72	**2.59**	6.94	2.86	21.38	**18.43**	413.54	A
12	**0.13**	**0.13**	0.36	0.18	0.59	**0.52**	1.73	0.65	2.65	**2.59**	6.80	2.84	21.89	**18.22**	411.58	A
13	**0.15**	0.17	0.38	0.22	**0.64**	0.70	1.80	0.79	**2.87**	3.64	7.24	3.39	**22.20**	28.31	432.33	A
Geometric Mean Normalized to DuckDB	**1.00**	1.69	1.80	1.13	1.00	1.53	1.98	**0.98**	1.00	1.69	2.17	**0.94**	**1.00**	1.48	8.74	A

Evaluation – Thin Variant



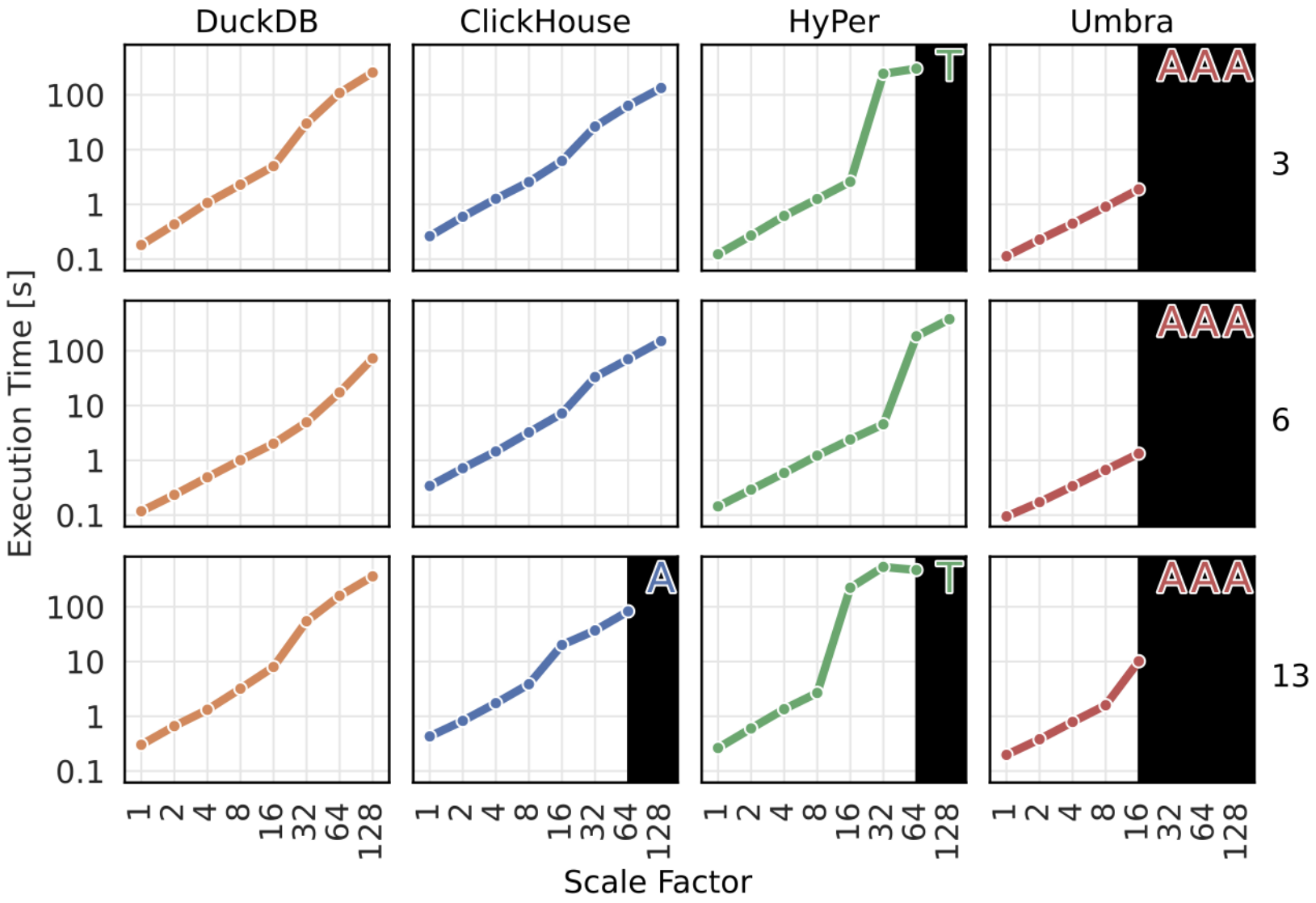
Evaluation – Wide Variant

TABLE III

EXECUTION TIME IN SECONDS FOR THE WIDE VARIANT OF ALL GROUPINGS AT SCALE FACTORS 2, 8, 32, AND 128. LOWER IS BETTER. THE LOWEST EXECUTION TIMES ARE HIGHLIGHTED IN BOLD. 'A' DENOTES THAT THE QUERY WAS ABORTED. 'T' DENOTES THAT THE QUERY TIMED OUT AFTER 600 SECONDS. WE SUMMARIZE BY NORMALIZING EXECUTION TIMES TO DUCKDB AND THEN TAKING THE GEOMETRIC MEAN.

SF	2				8				32				128			
System	Du	Cl	Hy	Um	Du	Cl	Hy	Um	Du	Cl	Hy	Um	Du	Cl	Hy	Um
Grouping 1	0.04	0.19	0.03	**0.02**	0.16	0.67	0.10	**0.06**	0.63	2.57	**0.38**	A	2.57	9.91	**1.52**	A
2	0.42	0.34	0.23	**0.22**	2.25	1.45	1.07	**0.77**	52.79	**18.41**	211.11	A	347.28	**111.66**	499.04	A
3	0.43	0.59	0.27	**0.23**	2.30	2.56	1.25	**0.91**	30.00	**26.39**	243.30	A	256.29	**133.50**	T	A
4	0.53	0.51	0.27	**0.23**	2.77	2.43	1.28	**0.91**	53.46	**26.19**	255.55	A	350.96	**122.97**	T	A
5	0.25	0.58	0.23	**0.13**	0.80	2.96	0.89	**0.44**	4.63	30.16	**3.13**	A	**67.56**	A	287.87	A
6	0.23	0.72	0.29	**0.17**	1.01	3.24	1.22	**0.67**	4.96	33.20	**4.56**	A	**72.69**	150.75	378.23	A
7	0.51	0.75	0.35	**0.29**	2.56	3.34	1.76	**1.22**	**30.61**	31.85	382.80	A	**260.12**	A	T	A
8	0.87	1.01	0.42	**0.36**	3.94	4.57	1.96	**1.47**	68.49	**38.79**	487.08	A	**407.26**	A	T	A
9	0.59	0.88	0.47	**0.32**	3.22	4.24	2.18	**1.41**	46.12	**36.37**	451.81	A	**331.08**	A	T	T
10	0.75	1.00	0.61	**0.42**	3.69	4.91	2.69	**1.74**	64.84	**43.70**	585.54	A	**399.46**	A	T	A
11	0.64	0.83	0.61	**0.38**	3.22	3.75	2.71	**1.60**	60.04	**36.41**	530.67	A	**396.53**	A	T	A
12	0.62	0.79	0.62	**0.38**	3.32	3.76	2.70	**1.58**	60.73	**36.00**	533.54	A	**382.09**	A	T	A
13	0.66	0.83	0.60	**0.38**	3.20	3.85	2.67	**1.59**	54.55	**37.00**	534.56	A	**359.95**	A	T	A
Geometric Mean Normalized to DuckDB	1.00	1.53	0.77	**0.54**	1.00	1.45	0.70	**0.45**	**1.00**	1.06	4.54	A	**1.00**	A	T	A

Evaluation – Wide Variant



Outline

Table data vs. temporary data

Buffer management for table data

- Page-based buffer management

Buffer management for temporary data

Unified memory management

Future work

Future Work

- Apply the idea to OLTP setting
- Better eviction policy for unified memory management
- **Adapt other blocking operators** besides hash aggregation
  - E.g., join, sort, window operators
- Coordinate when multiple memory-intensive operators are active at the same time

Questions – Buffer Management

- How common do real-world intermediates exceed memory?
- Challenges in extending to join, sorts, etc.?
- Multi-tenancy? (budget memory between simultaneous operators)
- Effective on skewed data?
- Row-major format for temporary tables? Good idea?
- How does the idea extend to distributed system?

Discussion

Problem: How to manage spilling for temporary data

DuckDB approach: Page-based data structures as much as possible
- **Benefit**: Data spilling is implicitly handled by buffer manager
- **Challenge**: Nontrivial for certain data structures and operators

Traditional approach: Use malloc for temporary data
- **Benefit**: intuitive data structures and operators
- **Challenge**: Data spilling must be explicitly handled

Discussion question: Is there a framework that allows intuitive data structures & operators and yet supports efficient spilling?

Before Next Lecture

Submit review for

David DeWitt, Jim Gray, Parallel Database Systems: The Future of High Performance Database Processing. Communications of the ACM, 1992