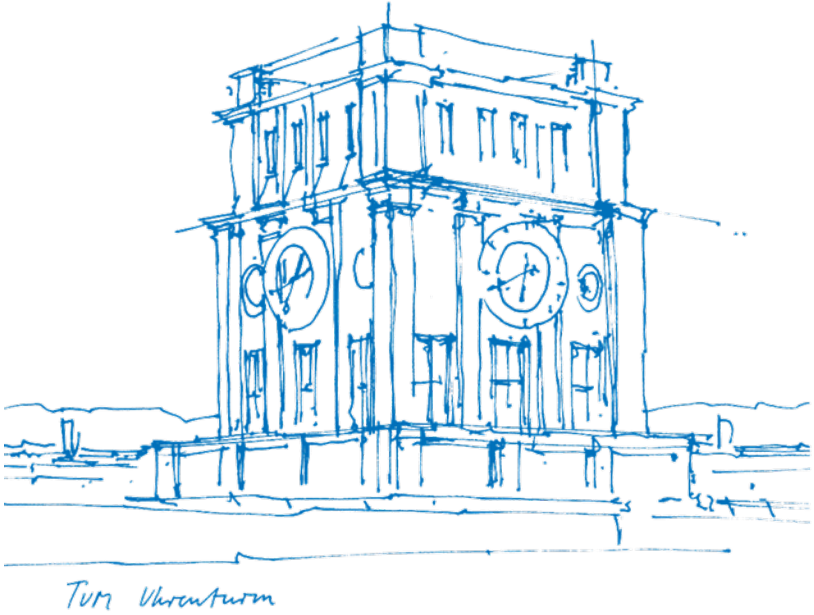
Building Advanced Analytics from Low-Level Plan Operators

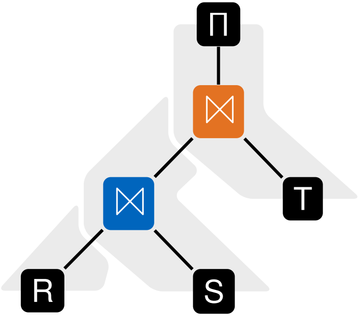
André Kohn, Viktor Leis, Thomas Neumann

Technical University Munich

SIGMOD 2021



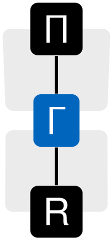
Motivation




```
for (a,b) in R:
  ht1.insert(a,b)

for (c,d) in S:
  for (a,b) in ht1.lookup(d):
    ht2.insert(c,(a,b))

for e in T:
  for (c,(a,b)) in ht2.lookup(e):
    print(a,b,c)
```

SELECT a, b, c **FROM** R, S, T **WHERE** a = d **AND** c = e

Motivation




```
for (a,b,c,d) in R:
  partitions.insert(d,(a,b))
  agg1.preagg((d,c),())
```

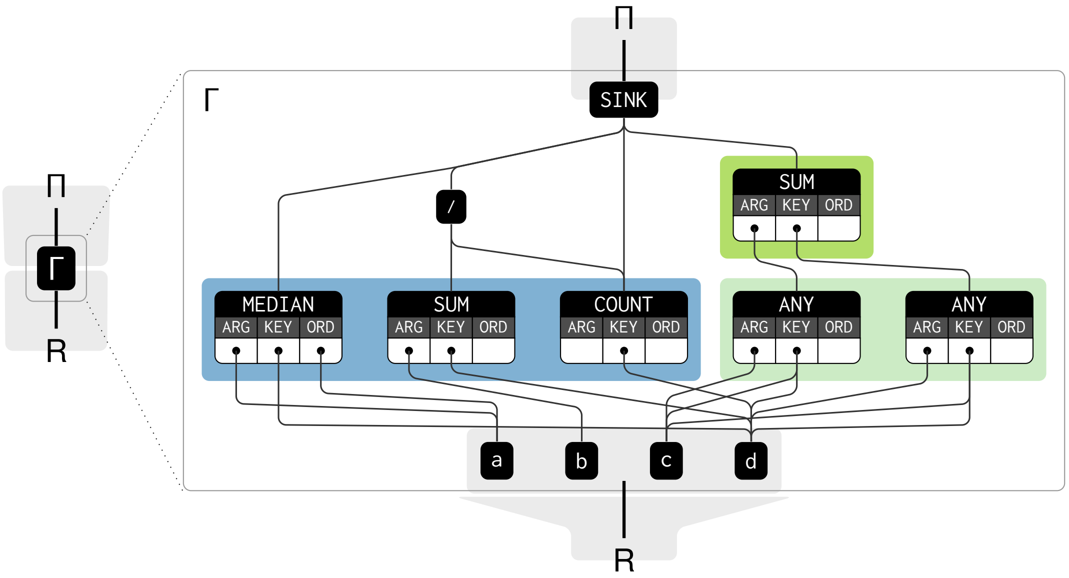
```
partitions.shuffle()
partitions.sort((d,a))
for (md,sum,cnt) in partitions:
  ht2[d] = (md,sum,cnt,NULL)
for (d,c) in agg1.merge():
  agg2.preagg(d, sum(c))
for (d,sumc) in agg2.merge():
  ht2[d][3] = sumc
```

```
for (d,md,sum,cnt,sumc) in ht2:
  print(d,md,sum/cnt,sumc)
```

SELECT median(a), avg(b), sum(**DISTINCT** c) **FROM** R **GROUP BY** d

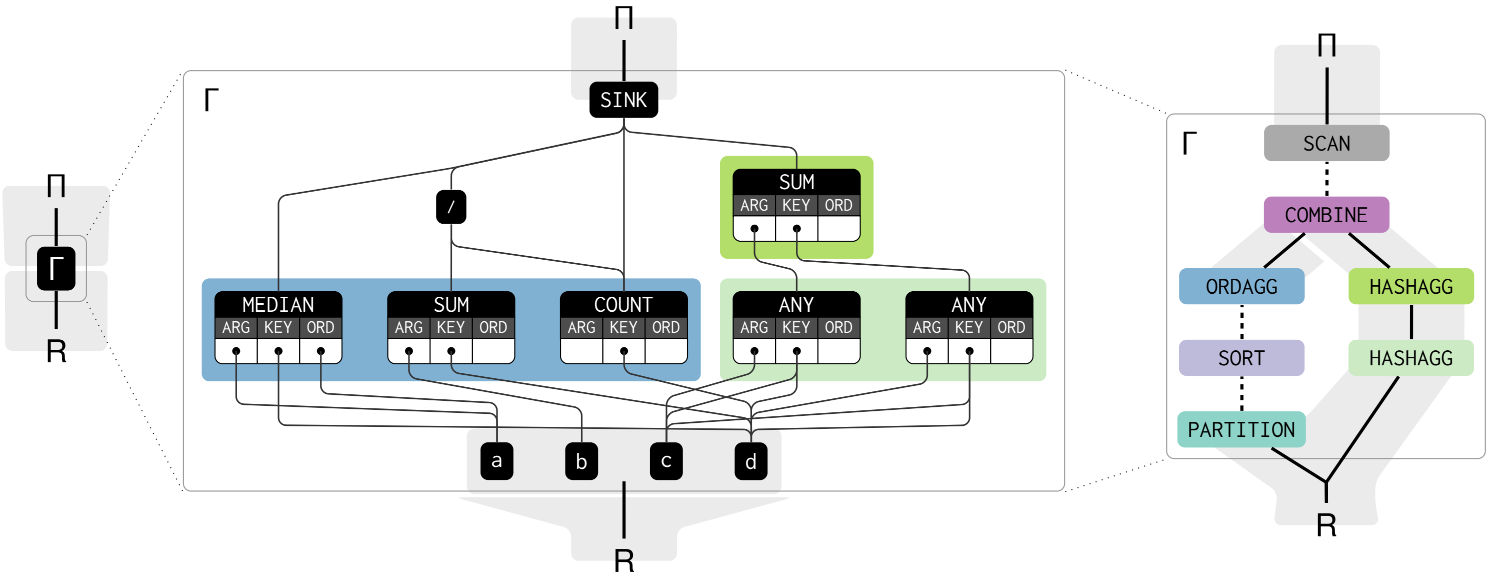
Motivation

SQL: `SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d`



Motivation

SQL: SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d



Low-Level Plan Operators

Problem:

- Relational Algebra favors monolithic aggregation logic.
- Set semantics prevent modular aggregation operators.
- Query plans should be DAGs rather than trees.

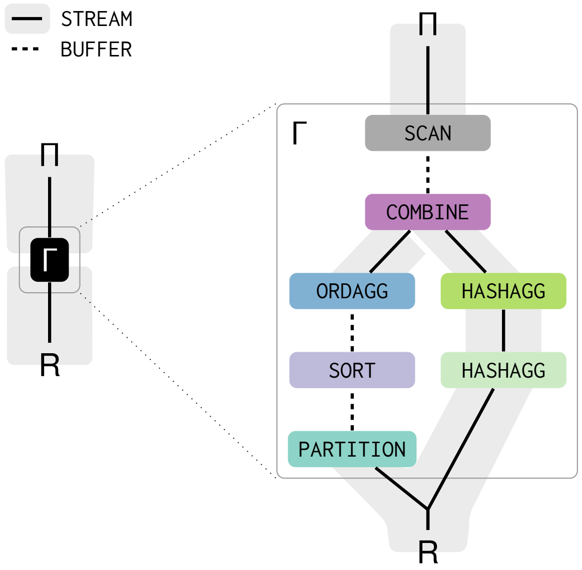
Solution:

- Introduce Low-Level-Plan Operators that consume and produce tuples with *Physical Properties*.
- Tuples can be *streamed*. (in , out)
- Tuples can be *materialized*. (in , out)
- Tuples can be *partitioned* and *ordered*.

	Operator	In	Out	Semantics
Transform	PARTITION			Hash-partitions input
	SORT			Sorts hash partitions
	MERGE			Merges hash partitions
	COMBINE			Joins unique groups
	SCAN			Scans hash partitions
Compute	HASHAGG			Aggregates hash-based
	ORDAGG			Aggregates sort-based
	WINDOW			Aggregates windows
	*			Traditional operators

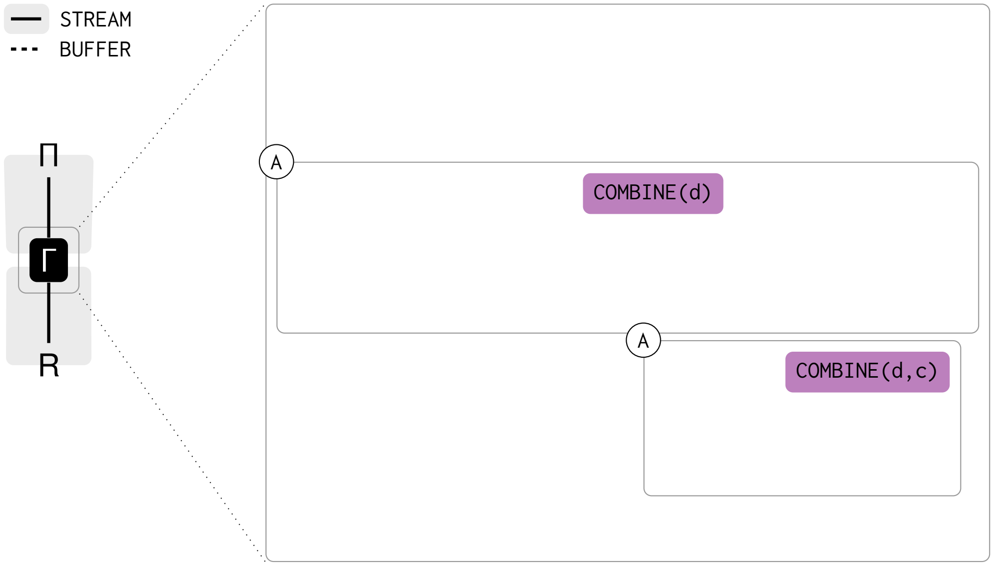
From Tree To DAG

SQL: SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d



From Tree To DAG

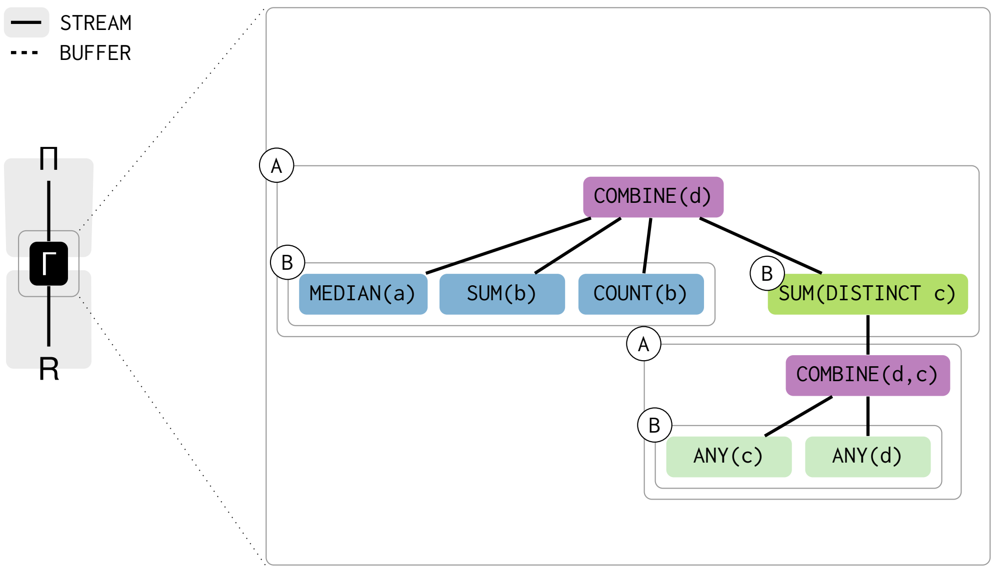
SQL: SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d



Ⓐ Add combine operators

From Tree To DAG

SQL: SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d



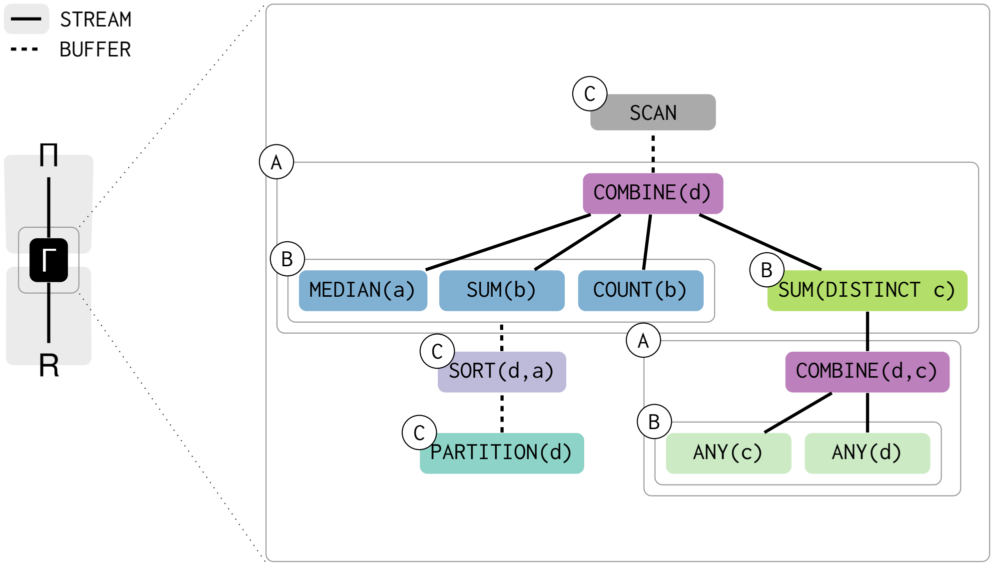
Ⓐ Add combine operators

Ⓑ Compute aggregates

- Expand grouping sets
- Select aggregation order
- Select aggregation strategies

From Tree To DAG

SQL: SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d



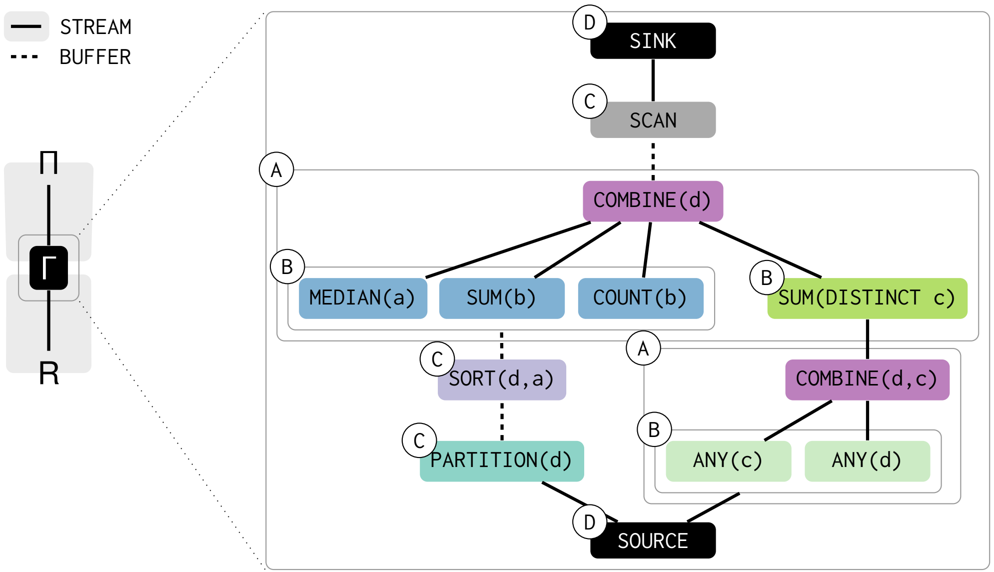
Ⓐ Add combine operators

Ⓑ Compute aggregates
- Expand grouping sets
- Select aggregation order
- Select aggregation strategies

Ⓒ Propagate buffers
- Add sorting operators
- Add partitioning operators
- Add scan operators

From Tree To DAG

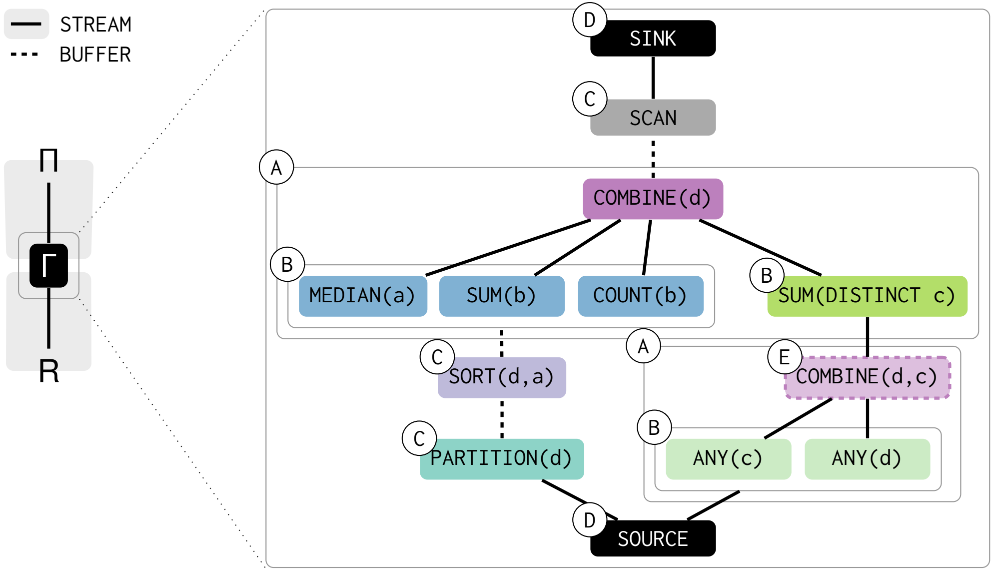
SQL: `SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d`



- Ⓐ Add combine operators
- Ⓑ Compute aggregates
  - Expand grouping sets
  - Select aggregation order
  - Select aggregation strategies
- Ⓒ Propagate buffers
  - Add sorting operators
  - Add partitioning operators
  - Add scan operators
- Ⓓ Connect DAG
  - Consume from source operator
  - Produce for sink operator

From Tree To DAG

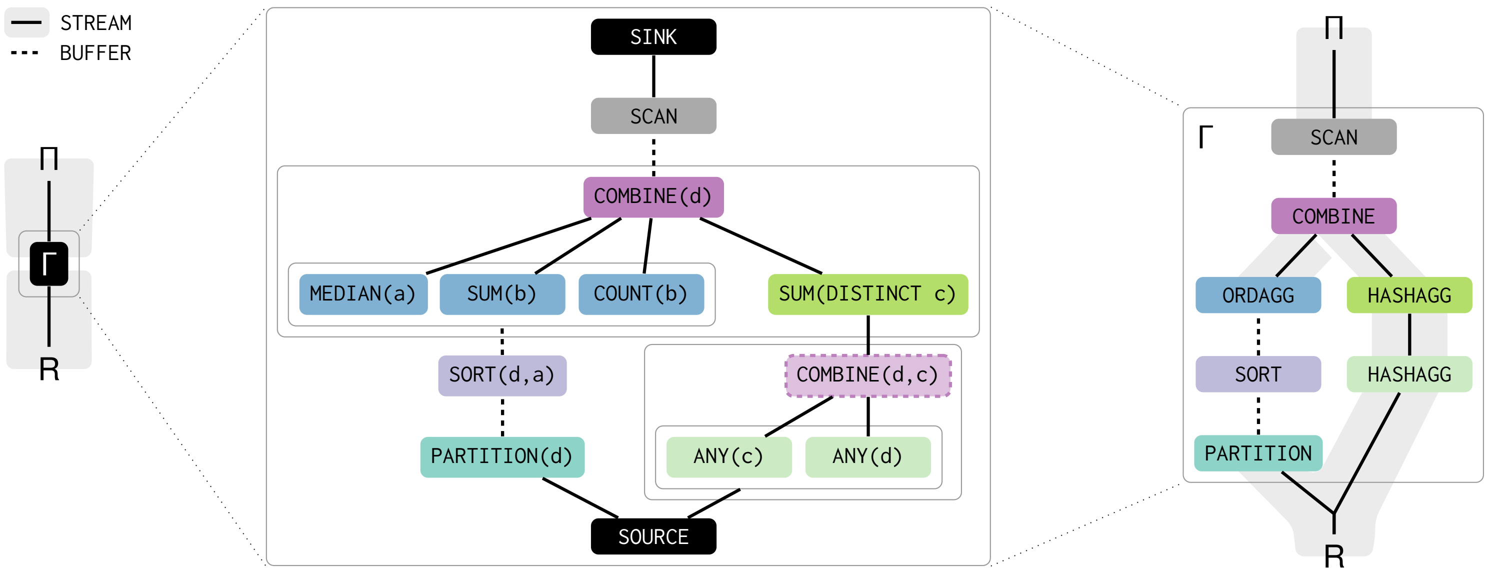
SQL: `SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d`



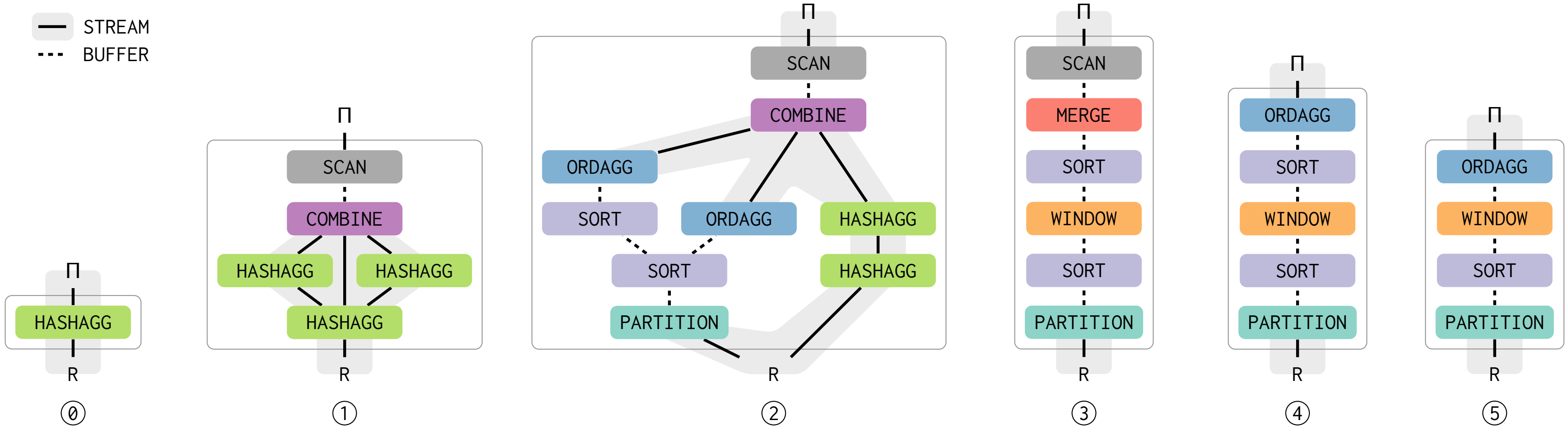
- (A) Add combine operators
- (B) Compute aggregates
  - Expand grouping sets
  - Select aggregation order
  - Select aggregation strategies
- (C) Propagate buffers
  - Add sorting operators
  - Add partitioning operators
  - Add scan operators
- (D) Connect DAG
  - Consume from source operator
  - Produce for sink operator
- (E) Optimize DAG
  - Replace unbounded windows
  - Remove redundant combines
  - Select producer order
  - Select buffer layouts
  - Select sort modes

From Tree To DAG

SQL: `SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d`

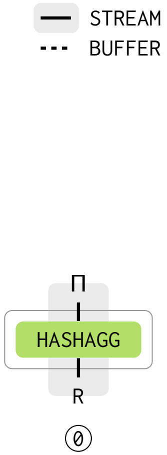


Advanced Aggregates



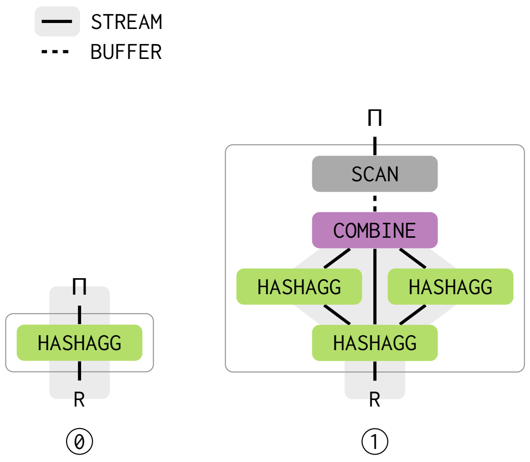

```
⓪ SELECT a, var_pop(b), count(b), sum(b) FROM R GROUP BY a
① SELECT a, b, sum(c) FROM R GROUP BY GROUPING SETS ((a), (b), (a, b))
② SELECT a, sum(b), sum(DISTINCT b), percentile_disc(0.5) WITHIN GROUP (ORDER BY c),
   percentile_disc(0.5) WITHIN GROUP (ORDER BY d) FROM R GROUP BY a
③ SELECT row_number() OVER (PARTITION BY a ORDER BY b) FROM R ORDER BY c LIMIT 100
④ SELECT a, mad() WITHIN GROUP (ORDER BY b) FROM R GROUP BY a
⑤ SELECT b, sum(pow(next_a - a, 2)) / nullif(count(*) - 1, 0)
  FROM (SELECT b, a, lead(a) OVER (PARTITION BY b ORDER BY a) AS next_a FROM R GROUP BY b)
```

Advanced Aggregates



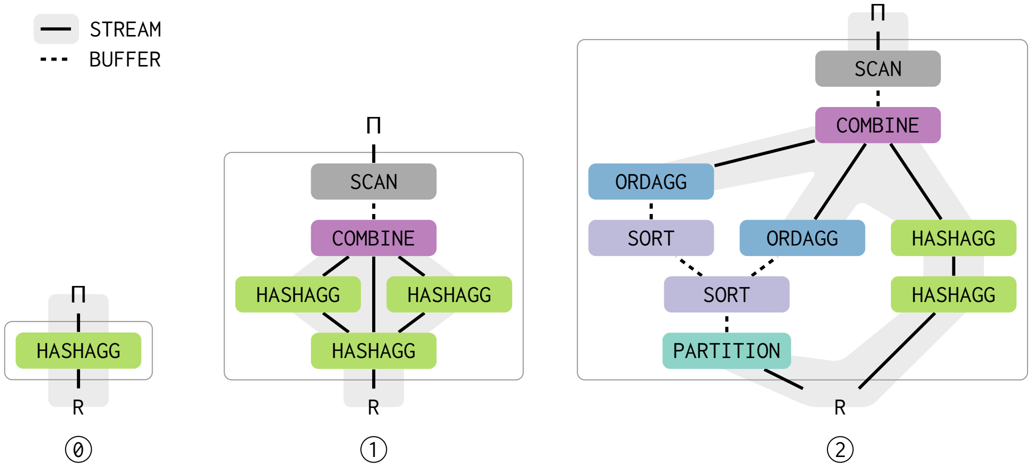
0 **SELECT** a, var_pop(b), count(b), sum(b) **FROM** R **GROUP BY** a

Advanced Aggregates



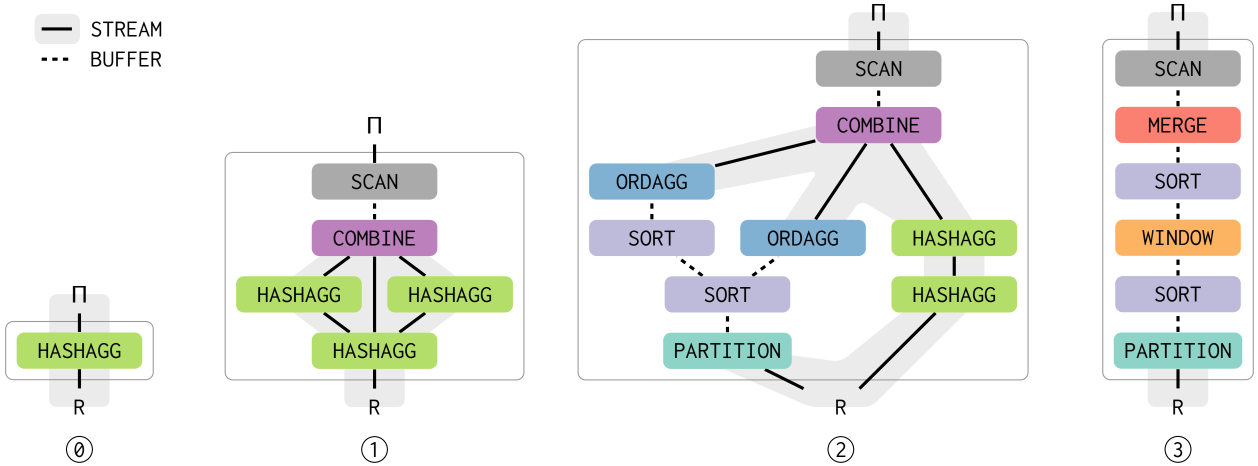

```
⓪ SELECT a, var_pop(b), count(b), sum(b) FROM R GROUP BY a
① SELECT a, b, sum(c) FROM R GROUP BY GROUPING SETS ((a), (b), (a, b))
```

Advanced Aggregates



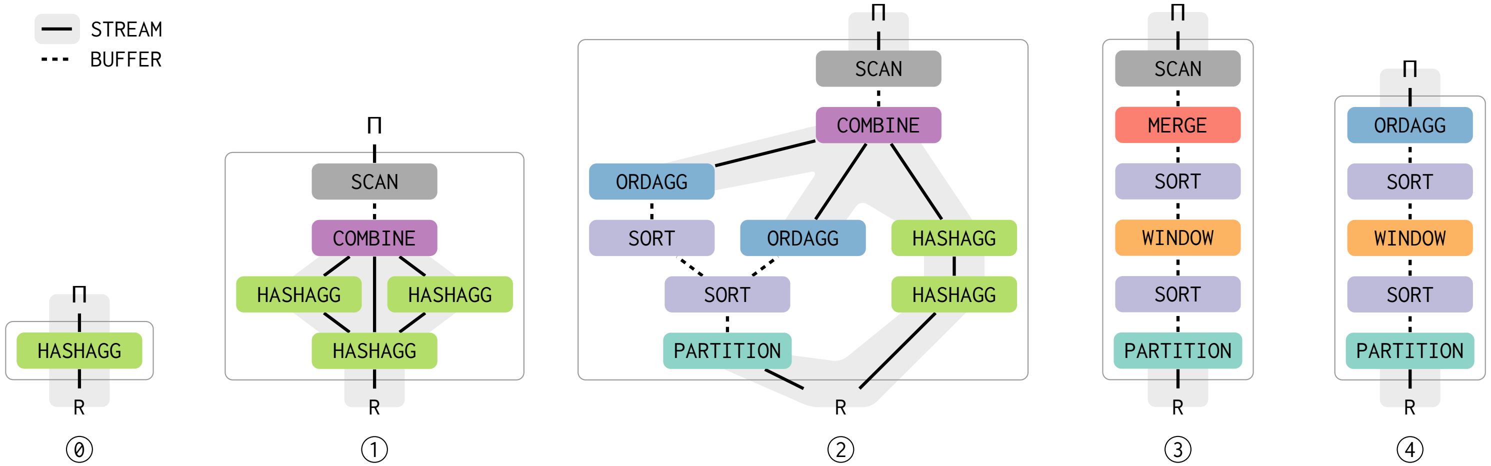
⓪ **SELECT** a, var_pop(b), count(b), sum(b) **FROM** R **GROUP BY** a
① **SELECT** a, b, sum(c) **FROM** R **GROUP BY GROUPING SETS** ((a), (b), (a, b))
② **SELECT** a, sum(b), sum(**DISTINCT** b), percentile_disc(0.5) **WITHIN GROUP** (**ORDER BY** c),
percentile_disc(0.5) **WITHIN GROUP** (**ORDER BY** d) **FROM** R **GROUP BY** a

Advanced Aggregates



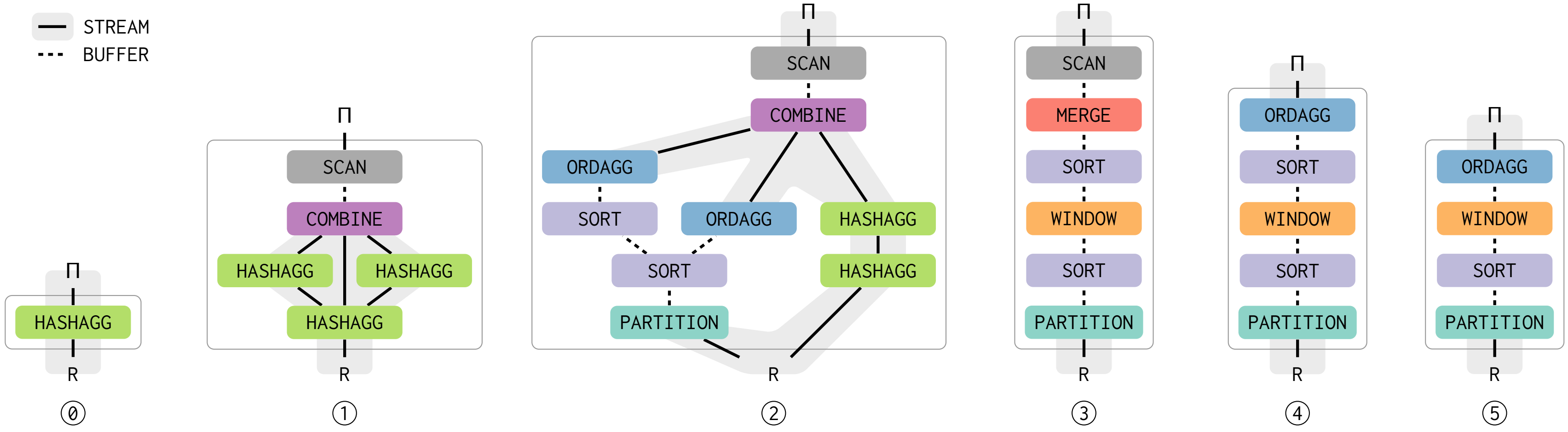
⓪ **SELECT** a, var_pop(b), count(b), sum(b) **FROM** R **GROUP BY** a
① **SELECT** a, b, sum(c) **FROM** R **GROUP BY GROUPING SETS** ((a), (b), (a, b))
② **SELECT** a, sum(b), sum(**DISTINCT** b), percentile_disc(0.5) **WITHIN GROUP** (**ORDER BY** c), percentile_disc(0.5) **WITHIN GROUP** (**ORDER BY** d) **FROM** R **GROUP BY** a
③ **SELECT** row_number() **OVER** (**PARTITION BY** a **ORDER BY** b) **FROM** R **ORDER BY** c **LIMIT** 100

Advanced Aggregates




```
⓪ SELECT a, var_pop(b), count(b), sum(b) FROM R GROUP BY a
① SELECT a, b, sum(c) FROM R GROUP BY GROUPING SETS ((a), (b), (a, b))
② SELECT a, sum(b), sum(DISTINCT b), percentile_disc(0.5) WITHIN GROUP (ORDER BY c),
   percentile_disc(0.5) WITHIN GROUP (ORDER BY d) FROM R GROUP BY a
③ SELECT row_number() OVER (PARTITION BY a ORDER BY b) FROM R ORDER BY c LIMIT 100
④ SELECT a, mad() WITHIN GROUP (ORDER BY b) FROM R GROUP BY a
```

Advanced Aggregates

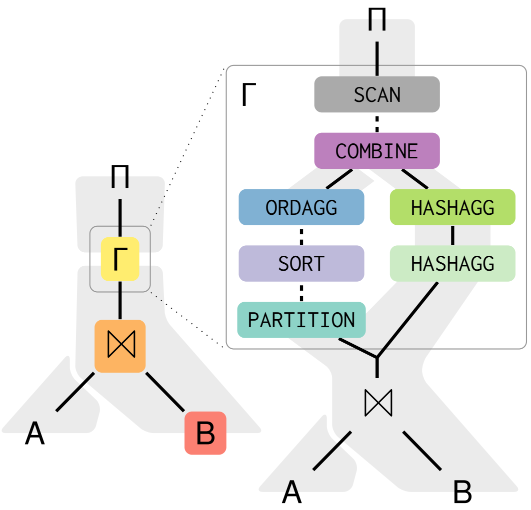



```
⓪ SELECT a, var_pop(b), count(b), sum(b) FROM R GROUP BY a
① SELECT a, b, sum(c) FROM R GROUP BY GROUPING SETS ((a), (b), (a, b))
② SELECT a, sum(b), sum(DISTINCT b), percentile_disc(0.5) WITHIN GROUP (ORDER BY c),
   percentile_disc(0.5) WITHIN GROUP (ORDER BY d) FROM R GROUP BY a
③ SELECT row_number() OVER (PARTITION BY a ORDER BY b) FROM R ORDER BY c LIMIT 100
④ SELECT a, mad() WITHIN GROUP (ORDER BY b) FROM R GROUP BY a
⑤ SELECT b, sum(pow(next_a - a, 2)) / nullif(count(*) - 1, 0)
  FROM (SELECT b, a, lead(a) OVER (PARTITION BY b ORDER BY a) AS next_a FROM R GROUP BY b)
```

Code Generation

SQL: SELECT median(a), avg(b), sum(DISTINCT c) FROM A, B WHERE e = f GROUP BY d

```
for e in A:
  ht1.insert(e)

for (a,b,c,d,f) in B:
  for e in ht1.lookup(f):
    partitions.insert(d,(a,b))
    agg1.preagg((d,c),())

partitions.shuffle()
partitions.sort((d,a))
for (md,sum,cnt) in partitions:
  ht2[d] = (md,sum,cnt,NULL)
for (d,c) in agg1.merge():
  agg2.preagg(d, sum(c))
for (d,sumc) in agg2.merge():
  ht2[d][3] = sumc

for (d,md,sum,cnt,sumc) in ht2:
  print(d,md,sum/cnt,sumc)
```

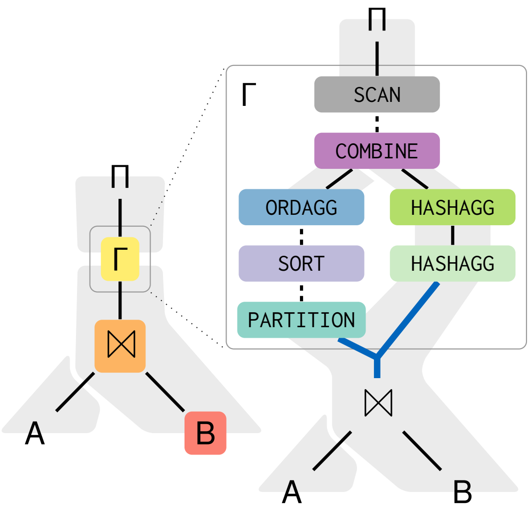



```
for e in A:
  ht1.insert(e)

for (a,b,c,d,f) in B:
  for e in ht1.lookup(f):
    partitions.insert(d,(a,b))
    agg1.preagg((d,c),())

partitions.shuffle()
partitions.sort((d,a))

for (md,sum,cnt) in partitions:
  ht2[d] = (md,sum/cnt,NULL)

for (d,c) in agg1.merge():
  agg2.preagg(d, sum(c))

for (d,sumc) in agg2.merge():
  ht2[d][3] = sumc

for (d,md,avg,sumc) in ht2:
  print(d,md,avg,sumc)
```

Code Generation

SQL: SELECT median(a), avg(b), sum(DISTINCT c) FROM A, B WHERE e = f GROUP BY d

```
for e in A:
  ht1.insert(e)

for (a,b,c,d,f) in B:
  for e in ht1.lookup(f):
    partitions.insert(d,(a,b))
    agg1.preagg((d,c),())

partitions.shuffle()
partitions.sort((d,a))
for (md,sum,cnt) in partitions:
  ht2[d] = (md,sum,cnt,NULL)
for (d,c) in agg1.merge():
  agg2.preagg(d, sum(c))
for (d,sumc) in agg2.merge():
  ht2[d][3] = sumc

for (d,md,sum,cnt,sumc) in ht2:
  print(d,md,sum/cnt,sumc)
```

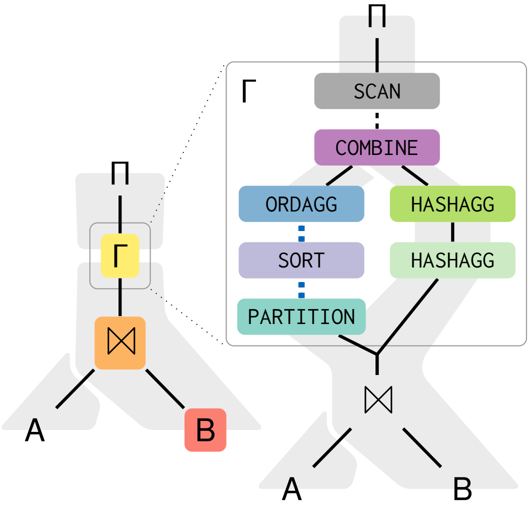



```
for e in A:
  ht1.insert(e)

for (a,b,c,d,f) in B:
  for e in ht1.lookup(f):
    partitions.insert(d,(a,b))
    agg1.preagg((d,c),())

partitions.shuffle()
partitions.sort((d,a))

for (md,sum,cnt) in partitions:
  ht2[d] = (md,sum/cnt,NULL)

for (d,c) in agg1.merge():
  agg2.preagg(d, sum(c))

for (d,sumc) in agg2.merge():
  ht2[d][3] = sumc

for (d,md,avg,sumc) in ht2:
  print(d,md,avg,sumc)
```

Code Generation

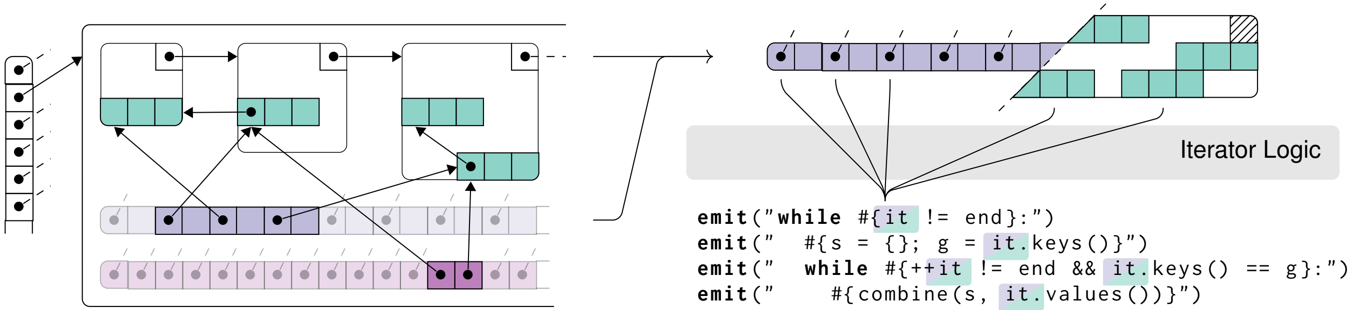
SQL: SELECT median(a), avg(b), sum(DISTINCT c) FROM A, B WHERE e = f GROUP BY d

```
for e in A:
  ht1.insert(e)

for (a,b,c,d,f) in B:
  for e in ht1.lookup(f):
    partitions.insert(d,(a,b))
    agg1.preagg((d,c),())

partitions.shuffle()
partitions.sort((d,a))
for (md,sum,cnt) in partitions:
  ht2[d] = (md,sum,cnt,NULL)
for (d,c) in agg1.merge():
  agg2.preagg(d, sum(c))
for (d,sumc) in agg2.merge():
  ht2[d][3] = sumc

for (d,md,sum,cnt,sumc) in ht2:
  print(d,md,sum/cnt,sumc)
```




```
for e in A:
  ht1.insert(e)

for (a,b,c,d,f) in B:
  for e in ht1.lookup(f):
    partitions.insert(d,(a,b))
    agg1.preagg((d,c),())

partitions.shuffle()
partitions.sort((d,a))

for (md,sum,cnt) in partitions:
  ht2[d] = (md,sum/cnt,NULL)

for (d,c) in agg1.merge():
  agg2.preagg(d, sum(c))

for (d,sumc) in agg2.merge():
  ht2[d][3] = sumc

for (d,md,avg,sumc) in ht2:
  print(d,md,avg,sumc)
```

Tuple Buffer

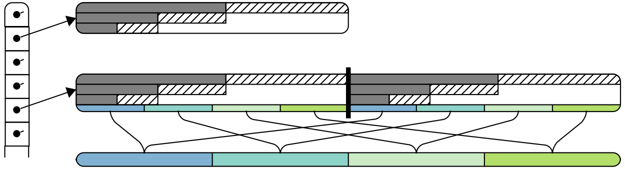



```
emit("while #{it != end}:")
emit("   #{s = {}; g = it.keys()}")
emit("   while #{++it != end && it.keys() == g}:")
emit("     #{combine(s, it.values())}")
emit("   #{consumer.consume(s)})
```

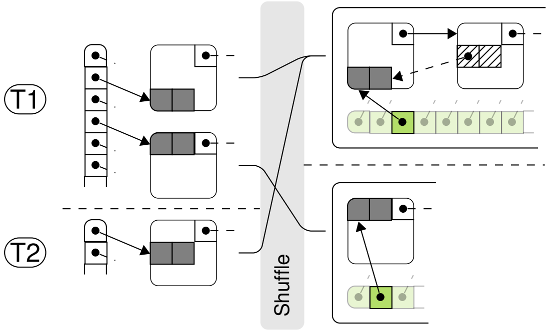
- Partitioned chunk lists for materialized tuples.
- Row-major layout for generated tuple access.
- Auxiliary *Permutation Vectors* and *Hash Tables*.
- Iterator abstraction during code generation.
- Sorting *in-place* or with *Permutation Vectors*.
- In-place sorting whenever tuples are small.

Implementation

SCAN	**Scans materialized hash partitions.**
PARTITION	**Materializes local hash partitions, merges across threads.**
MERGE	**Merges partitions with repeated parallel 64-way merges.**
SORT	Sorts partitions with a Morsel-Driven BlockQuicksort.
COMBINE	Builds partitioned hash tables after materializing input. Flushes missing groups to local hash partitions and rehashes between pipelines.

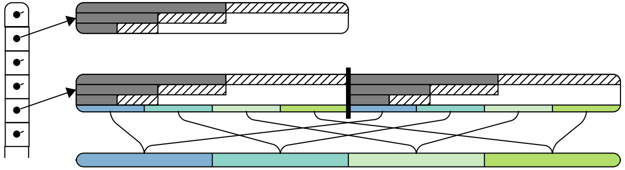


HASHAGG	Aggregates input in fixed-size local hash tables. Flushes collisions to hash partitions, then merges partial aggregates with dynamic tables.
ORDAGG	Aggregates sorted key ranges. Scans repeatedly for nested aggregates.
WINDOW	Evaluates multiple window frames for each row.

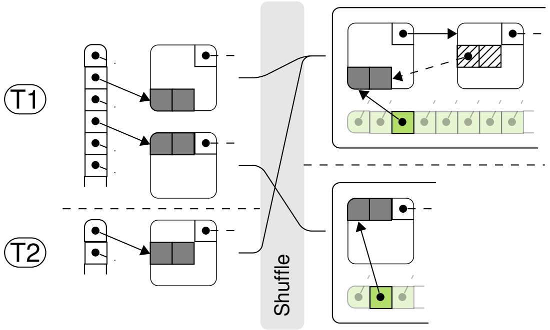


Implementation

`SCAN`	Scans materialized hash partitions.
`PARTITION`	Materializes local hash partitions, merges across threads.
`MERGE`	Merges partitions with repeated parallel 64-way merges.
`SORT`	**Sorts partitions with a Morsel-Driven BlockQuicksort.**
`COMBINE`	**Builds partitioned hash tables after materializing input.**
	Flushes missing groups to local hash partitions and rehashes between pipelines.

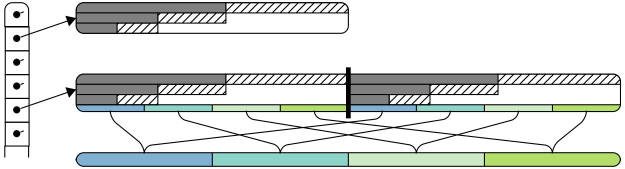


`HASHAGG`	Aggregates input in fixed-size local hash tables.
	Flushes collisions to hash partitions, then merges partial aggregates with dynamic tables.
`ORDAGG`	Aggregates sorted key ranges.
	Scans repeatedly for nested aggregates.
`WINDOW`	Evaluates multiple window frames for each row.

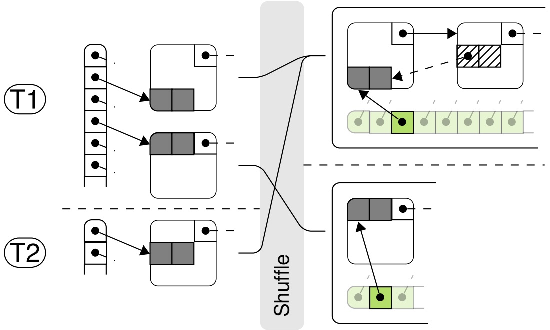


Implementation

SCAN	Scans materialized hash partitions.
PARTITION	Materializes local hash partitions, merges across threads.
MERGE	Merges partitions with repeated parallel 64-way merges.
SORT	Sorts partitions with a Morsel-Driven BlockQuicksort.
COMBINE	Builds partitioned hash tables after materializing input. Flushes missing groups to local hash partitions and rehashes between pipelines.

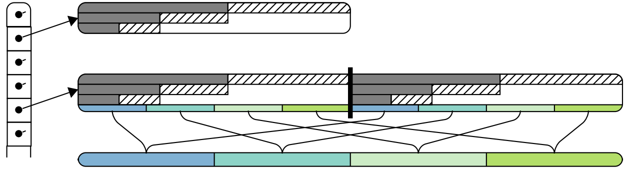


HASHAGG	**Aggregates input in fixed-size local hash tables. Flushes collisions to hash partitions, then merges partial aggregates with dynamic tables.**
ORDAGG	**Aggregates sorted key ranges. Scans repeatedly for nested aggregates.**
WINDOW	Evaluates multiple window frames for each row.

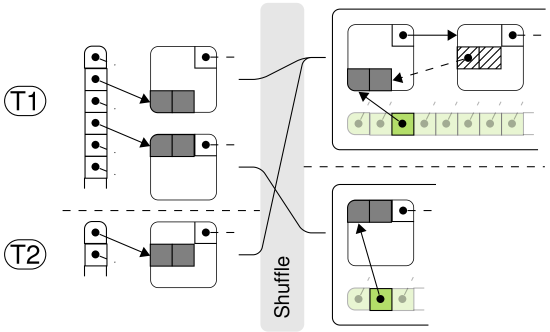


Implementation

`SCAN`	Scans materialized hash partitions.
`PARTITION`	Materializes local hash partitions, merges across threads.
`MERGE`	Merges partitions with repeated parallel 64-way merges.
`SORT`	Sorts partitions with a Morsel-Driven BlockQuicksort.
`COMBINE`	Builds partitioned hash tables after materializing input. Flushes missing groups to local hash partitions and rehashes between pipelines.



`HASHAGG`	Aggregates input in fixed-size local hash tables. Flushes collisions to hash partitions, then merges partial aggregates with dynamic tables.
`ORDAGG`	Aggregates sorted key ranges. Scans repeatedly for nested aggregates.
`WINDOW`	**Evaluates multiple window frames for each row.**



Versus HyPer

	#	Aggregates		1 thread Umbra	1 thread HyPer	1 thread ×	20 threads Umbra	20 threads HyPer	20 threads ×
Single	**1**	**`SUM(e), COUNT(e), VAR_SAMP(e)`**	**`GROUP BY k`**	**3.10**	**4.73**	**1.53**	**0.37**	**0.60**	**1.62**
	2	**`↳, PCTL(e,0.5)`**	**`GROUP BY k`**	**4.32**	**9.36**	**2.17**	**0.47**	**0.96**	**2.03**
	3	`COUNT(e), COUNT(DISTINCT e)`	`GROUP BY k`	9.61	127.63	13.28	1.21	26.52	21.90
Ordered-Set	4	`PCTL(e,0.5)`	`GROUP BY k`	4.00	8.88	2.22	0.43	0.92	2.14
	5	`↳, PCTL(e,0.99)`	`GROUP BY k`	4.02	12.66	3.15	0.42	1.40	3.31
	6	`↳, PCTL(q,0.5), PCTL(q,0.9)`	`GROUP BY k`	6.48	22.39	3.46	0.64	2.68	4.20
	7	`PCTL(e,0.5), PCTL(q,0.5)`	`GROUP BY n`	6.74	21.93	3.25	0.93	19.85	21.36
Grouping-Sets	8	`SUM(q)`	`GROUP BY ((k,n),(k),(n))`	2.30	10.73	4.66	0.28	1.09	3.96
	9	`SUM(q)`	`GROUP BY ((k,s,n),(k,s),(k,n),(n))`	2.63	16.37	6.22	0.42	1.71	4.09
	10	`PCTL(q,0.5)`	`GROUP BY ((k,n),(k))`	2.43	18.11	7.46	0.24	1.85	7.56
	11	`PCTL(q,0.5)`	`GROUP BY ((k,s,n),(k,s),(k))`	2.77	27.78	10.05	0.31	2.89	9.44
	12	`PCTL(q,0.5)`	`GROUP BY ((k,n),(k),(n))`	1.97	26.60	13.50	0.52	10.43	20.20
Window	13	`LEAD(q), LAG(q)`	`PARTITION BY k ORDER BY r`	8.33	13.69	1.64	0.97	1.46	1.50
	14	`↳, CUMSUM(q)`	`PARTITION BY k ORDER BY d`	12.77	19.05	1.49	1.56	2.27	1.46
	15	`CUMSUM(q)`	`PARTITION BY n ORDER BY d`	5.10	12.32	2.42	0.89	10.93	12.29
Nested	16	`PCTL(e - PCTL(e,0.5),0.5)`	`GROUP BY k`	6.35	12.39	1.95	0.69	1.44	2.07
	17	`PCTL(SUM(q), 0.5)`	`GROUP BY k`	1.58	4.08	2.58	0.20	0.52	2.62
	18	`SUM(POW(LEAD(q) - q,2)) / COUNT(*)`	`GROUP BY k`	5.63	10.90	1.94	0.58	1.09	1.89

`e=extendedprice` `n=linenumber` `s=linestatus` `o=orderkey` `p=partkey`
`q=quantity` `r=receiptdate` `k=suppkey` `d=shipdate` `m=shipmode`

Versus HyPer

	#	Aggregates		1 thread Umbra	1 thread HyPer	1 thread ×	20 threads Umbra	20 threads HyPer	20 threads ×
Single	1	SUM(e), COUNT(e), VAR_SAMP(e)	GROUP BY k	3.10	4.73	1.53	0.37	0.60	1.62
	2	↳, PCTL(e,0.5)	GROUP BY k	4.32	9.36	2.17	0.47	0.96	2.03
	3	COUNT(e), COUNT(DISTINCT e)	GROUP BY k	9.61	127.63	13.28	1.21	26.52	21.90
Ordered-Set	**4**	**PCTL(e,0.5)**	**GROUP BY k**	**4.00**	**8.88**	**2.22**	**0.43**	**0.92**	**2.14**
	5	**↳, PCTL(e,0.99)**	**GROUP BY k**	**4.02**	**12.66**	**3.15**	**0.42**	**1.40**	**3.31**
	6	**↳, PCTL(q,0.5), PCTL(q,0.9)**	**GROUP BY k**	**6.48**	**22.39**	**3.46**	**0.64**	**2.68**	**4.20**
	7	PCTL(e,0.5), PCTL(q,0.5)	GROUP BY n	6.74	21.93	3.25	0.93	19.85	21.36
Grouping-Sets	8	SUM(q)	GROUP BY ((k,n),(k),(n))	2.30	10.73	4.66	0.28	1.09	3.96
	9	SUM(q)	GROUP BY ((k,s,n),(k,s),(k,n),(n))	2.63	16.37	6.22	0.42	1.71	4.09
	10	PCTL(q,0.5)	GROUP BY ((k,n),(k))	2.43	18.11	7.46	0.24	1.85	7.56
	11	PCTL(q,0.5)	GROUP BY ((k,s,n),(k,s),(k))	2.77	27.78	10.05	0.31	2.89	9.44
	12	PCTL(q,0.5)	GROUP BY ((k,n),(k),(n))	1.97	26.60	13.50	0.52	10.43	20.20
Window	13	LEAD(q), LAG(q)	PARTITION BY k ORDER BY r	8.33	13.69	1.64	0.97	1.46	1.50
	14	↳, CUMSUM(q)	PARTITION BY k ORDER BY d	12.77	19.05	1.49	1.56	2.27	1.46
	15	CUMSUM(q)	PARTITION BY n ORDER BY d	5.10	12.32	2.42	0.89	10.93	12.29
Nested	16	PCTL(e - PCTL(e,0.5),0.5)	GROUP BY k	6.35	12.39	1.95	0.69	1.44	2.07
	17	PCTL(SUM(q), 0.5)	GROUP BY k	1.58	4.08	2.58	0.20	0.52	2.62
	18	SUM(POW(LEAD(q) - q,2)) / COUNT(*)	GROUP BY k	5.63	10.90	1.94	0.58	1.09	1.89

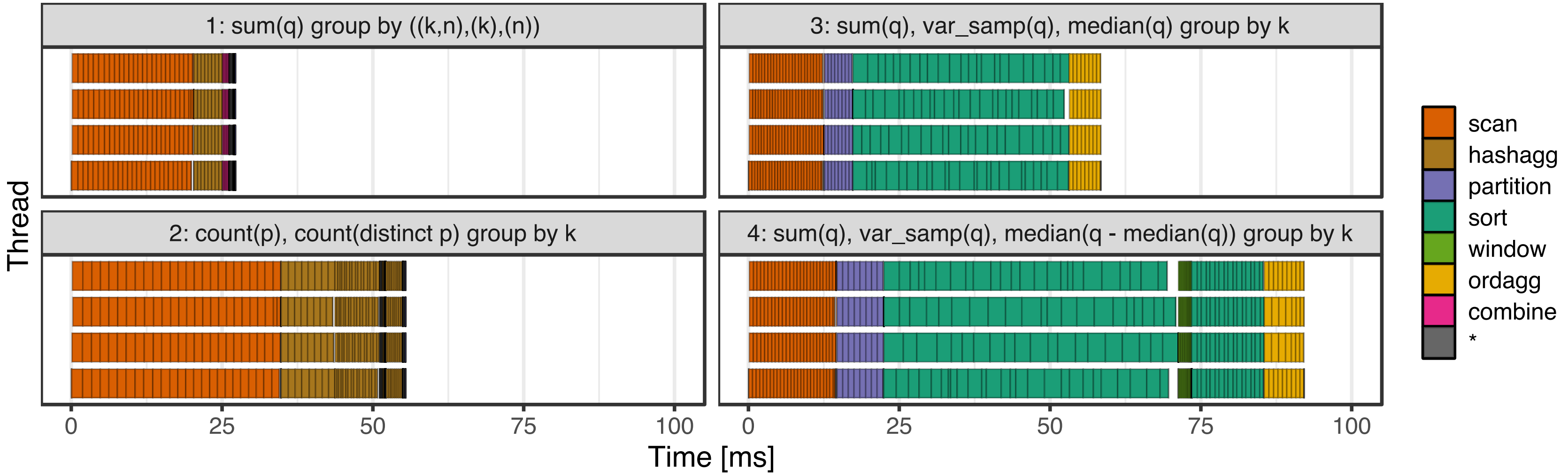
e=extendedprice n=linenumber s=linestatus o=orderkey p=partkey
q=quantity r=receiptdate k=suppkey d=shipdate m=shipmode

Versus HyPer

	#	Aggregates		1 thread Umbra	1 thread HyPer	1 thread ×	20 threads Umbra	20 threads HyPer	20 threads ×
Single	1	`SUM(e), COUNT(e), VAR_SAMP(e)`	`GROUP BY k`	3.10	4.73	1.53	0.37	0.60	1.62
	2	`↳, PCTL(e,0.5)`	`GROUP BY k`	4.32	9.36	2.17	0.47	0.96	2.03
	3	`COUNT(e), COUNT(DISTINCT e)`	`GROUP BY k`	9.61	127.63	13.28	1.21	26.52	21.90
Ordered-Set	4	`PCTL(e,0.5)`	`GROUP BY k`	4.00	8.88	2.22	0.43	0.92	2.14
	5	`↳, PCTL(e,0.99)`	`GROUP BY k`	4.02	12.66	3.15	0.42	1.40	3.31
	6	`↳, PCTL(q,0.5), PCTL(q,0.9)`	`GROUP BY k`	6.48	22.39	3.46	0.64	2.68	4.20
	7	`PCTL(e,0.5), PCTL(q,0.5)`	`GROUP BY n`	6.74	21.93	3.25	0.93	19.85	21.36
Grouping-Sets	8	`SUM(q)`	`GROUP BY ((k,n),(k),(n))`	2.30	10.73	4.66	0.28	1.09	3.96
	9	`SUM(q)`	`GROUP BY ((k,s,n),(k,s),(k,n),(n))`	2.63	16.37	6.22	0.42	1.71	4.09
	10	`PCTL(q,0.5)`	`GROUP BY ((k,n),(k))`	2.43	18.11	7.46	0.24	1.85	7.56
	11	`PCTL(q,0.5)`	`GROUP BY ((k,s,n),(k,s),(k))`	2.77	27.78	10.05	0.31	2.89	9.44
	12	`PCTL(q,0.5)`	`GROUP BY ((k,n),(k),(n))`	1.97	26.60	13.50	0.52	10.43	20.20
Window	13	`LEAD(q), LAG(q)`	`PARTITION BY k ORDER BY r`	8.33	13.69	1.64	0.97	1.46	1.50
	14	`↳, CUMSUM(q)`	`PARTITION BY k ORDER BY d`	12.77	19.05	1.49	1.56	2.27	1.46
	15	`CUMSUM(q)`	`PARTITION BY n ORDER BY d`	5.10	12.32	2.42	0.89	10.93	12.29
Nested	**16**	**`PCTL(e - PCTL(e,0.5),0.5)`**	**`GROUP BY k`**	**6.35**	**12.39**	**1.95**	**0.69**	**1.44**	**2.07**
	17	`PCTL(SUM(q), 0.5)`	`GROUP BY k`	1.58	4.08	2.58	0.20	0.52	2.62
	18	`SUM(POW(LEAD(q) - q,2)) / COUNT(*)`	`GROUP BY k`	5.63	10.90	1.94	0.58	1.09	1.89

e=extendedprice n=linenumber s=linestatus o=orderkey p=partkey
q=quantity r=receiptdate k=suppkey d=shipdate m=shipmode

In Action


1: sum(q) group by ((k,n),(k),(n))
3: sum(q), var_samp(q), median(q) group by k
2: count(p), count(distinct p) group by k
4: sum(q), var_samp(q), median(q - median(q)) group by k
Thread
0
25
50
75
100
Time [ms]
scan
hashagg
partition
sort
window
ordagg
combine
*

Low-Level Plan Operators modularize aggregation logic
and drive the efficient evaluation of **advanced** aggregation functions.