SASI, Cassandra on the full text search ride

DuyHai DOAN – Apache Cassandra evangelist

1	SASI introduction
2	SASI cluster-wide
3	SASI local read/write path
4	Query planner
5	Some benchmarks
6	Take away

SASI introduction

What is SASI ?

- **S**STable-**A**ttached **S**econdary **I**ndex → new 2nd index impl that follows SSTable life-cycle

- Objective: provide more performant & capable 2nd index

Who created it ?

Open-source contribution by an engineers team

Pavel Yaskevich
xedin

- Apple Inc.
- San Francisco, CA
- xedin@apache.org
- http://tinkerpop.com
- Joined on Aug 20, 2008

Jordan West
jrwest

- Apple
- San Francisco, CA
- jordanrw@gmail.com
- http://blog.jordanwest.me
- Joined on Dec 15, 2009

Jason Brown
jasobrown

- Apple
- Silicon Valley, CA
- http://www.apple.com
- Joined on Feb 1, 2012

Mikhail Stepura
Mishail

- Apple
- mikhail.stepura@outlook.com
- Joined on May 17, 2010

mkjellman

- Joined on May 11, 2012

Why is it better than native 2nd index ?

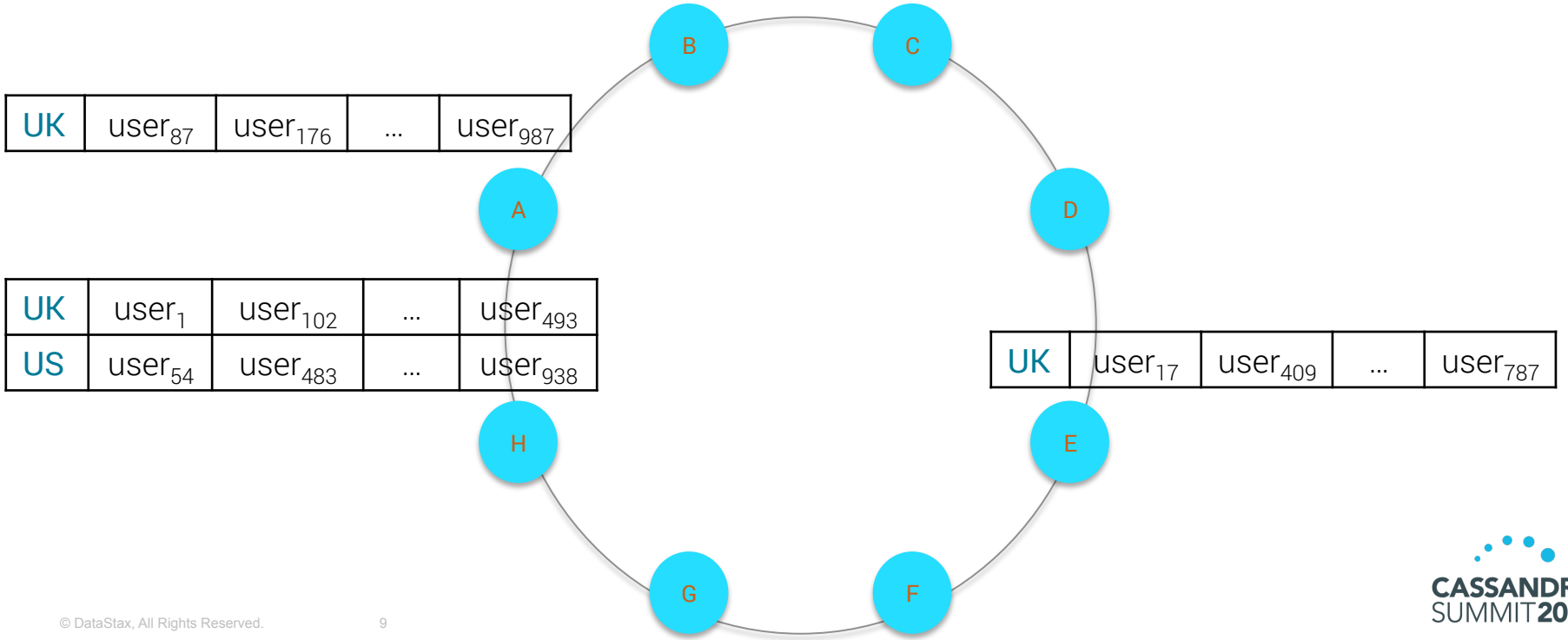
- follow SSTable life-cycle (flush, compaction, rebuild …) → more optimized
- new data-strutures
- range query (<, ≤, >, ≥) possible
- full text search options

Demo

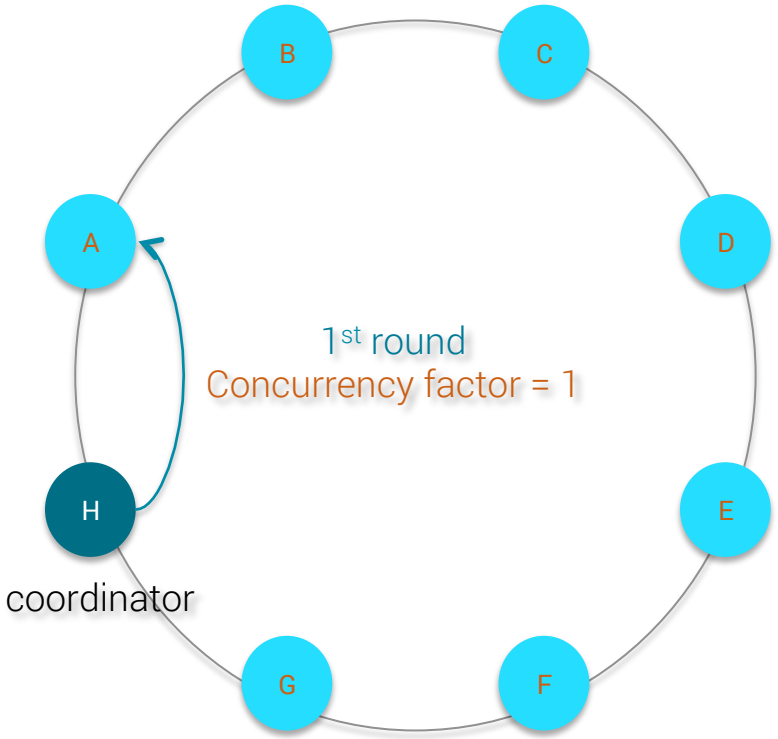
SASI cluster-wide

Distributed index

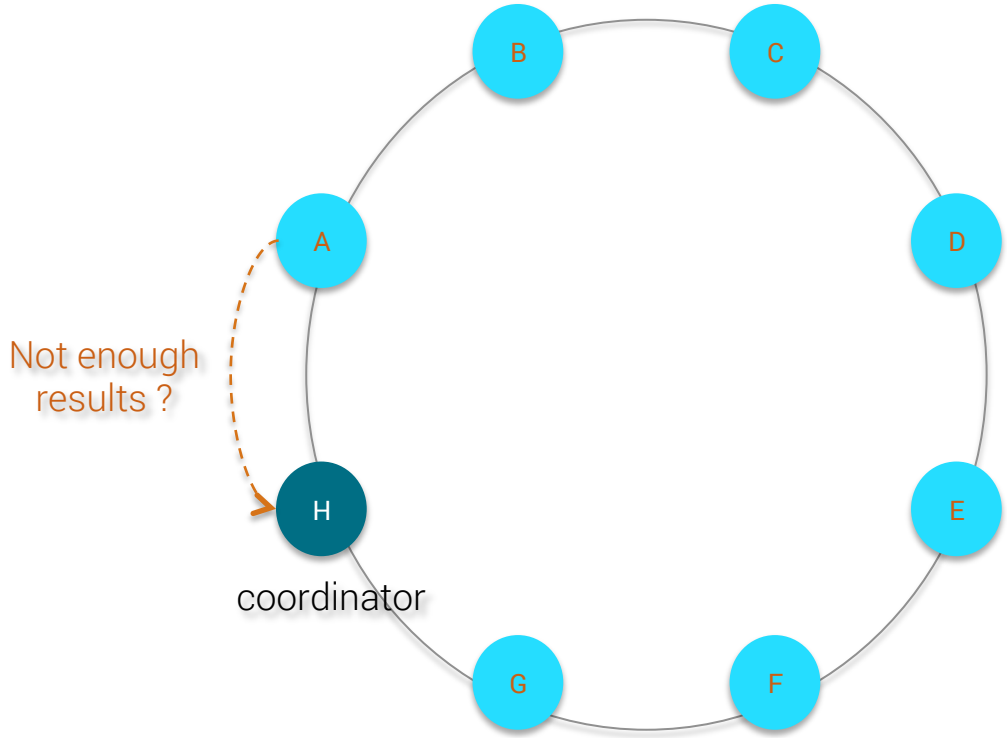
On cluster level, SASI works exactly like native 2nd index



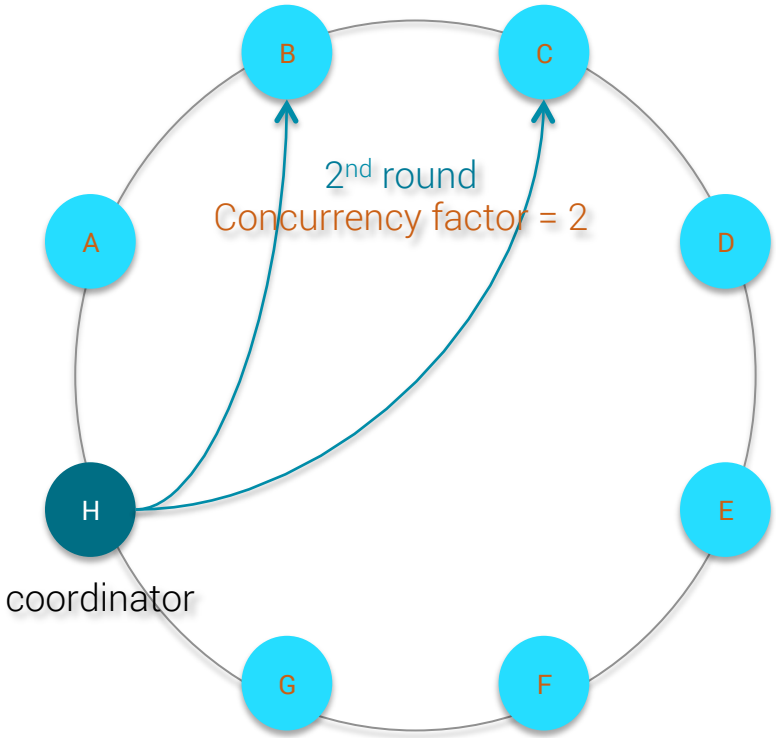
Distributed search algorithm



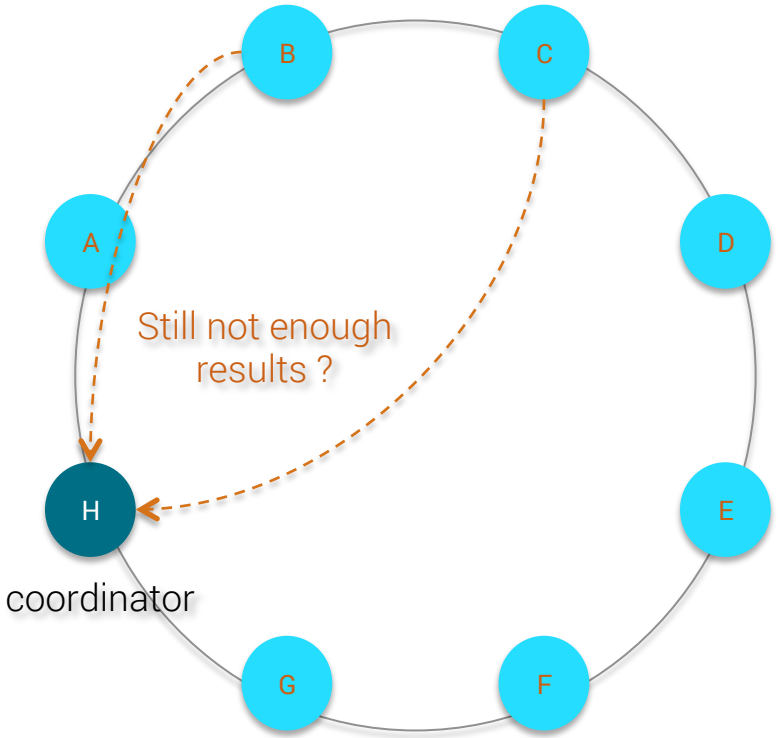
Distributed search algorithm



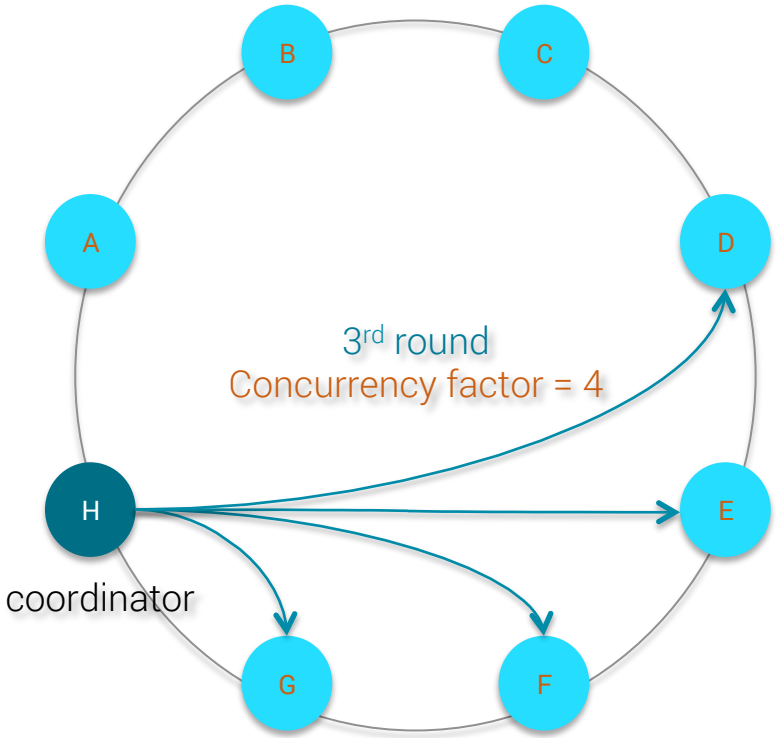
Distributed search algorithm



Distributed search algorithm



Distributed search algorithm



Concurrency factor formula

$$\text{CONCURRENCY\_FACTOR} = \max\left(1, \min\left(token\_range\_count, \left\lceil \frac{requested\_LIMIT}{estimate\_rows\_by\_token\_range} \right\rceil\right)\right)$$

$$estimate\_rows\_by\_token\_range = \frac{estimate\_rows}{token\_range\_count \times replication\_factor}$$

- more details at: http://www.planetcassandra.org/blog/cassandra-native-secondary-index-deep-dive/

Concurrency factor formula

But right now …

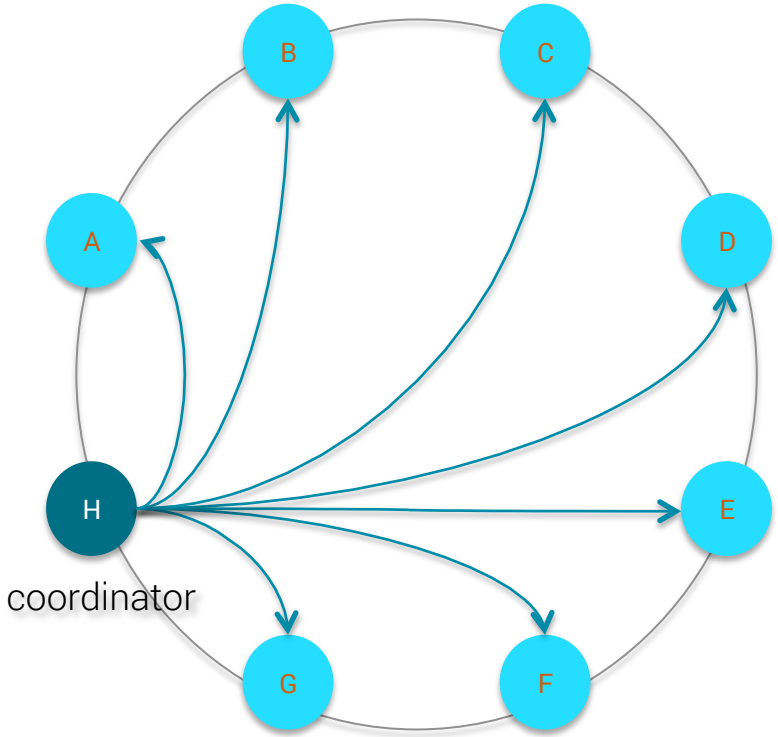
SASIIndex.getEstimatedResultRows()

```
1  public long getEstimatedResultRows()
2  {
3      // this is temporary (until proper QueryPlan is integrated into Cassandra)
4      // and allows us to priority SASI indexes if any in the query since they
5      // are going to be more efficient, to query and intersect, than built-in indexes.
6      return Long.MIN_VALUE;
7  }
```

So initial concurrency factor always = *max(1, negative number)* = 1 for 1st query round with SASI …

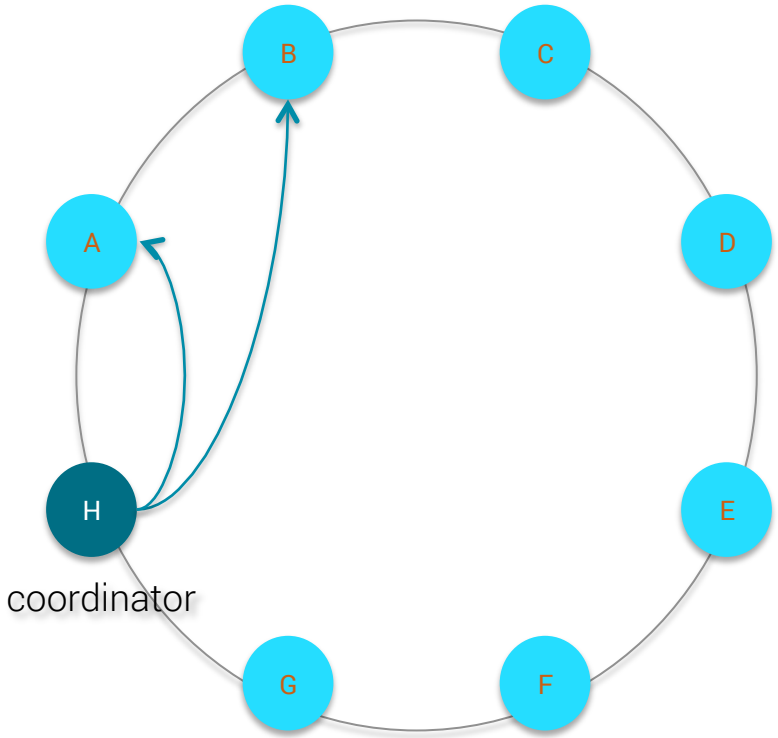
Caveat 1: non restrictive filters

Hit all nodes eventually ☹

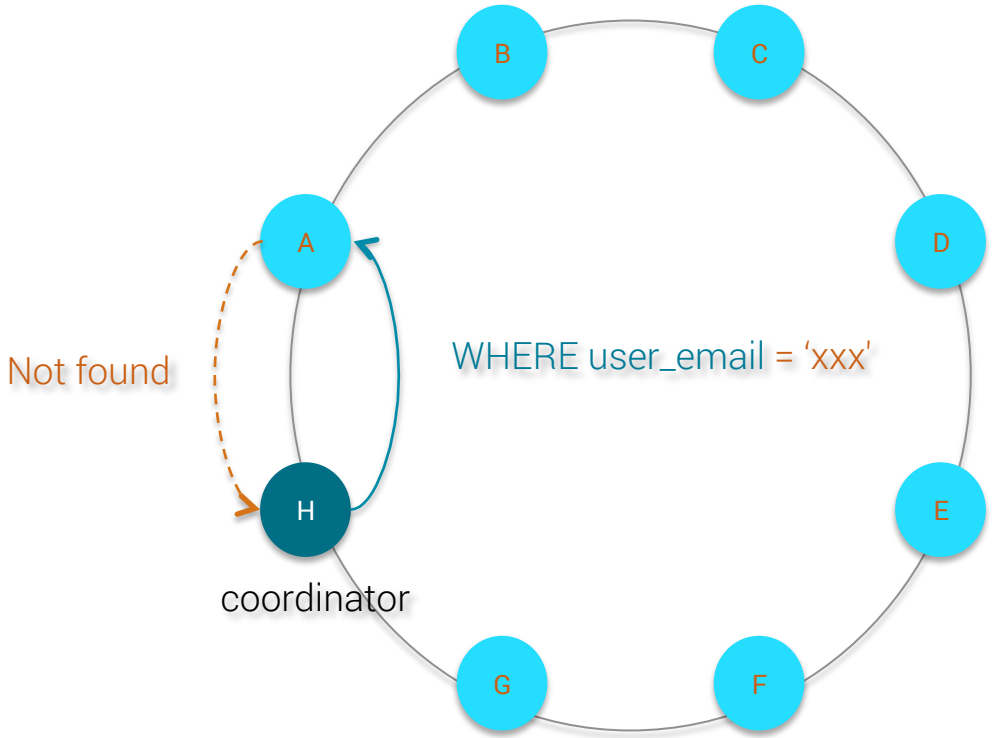


Caveat 1 solution : always use LIMIT

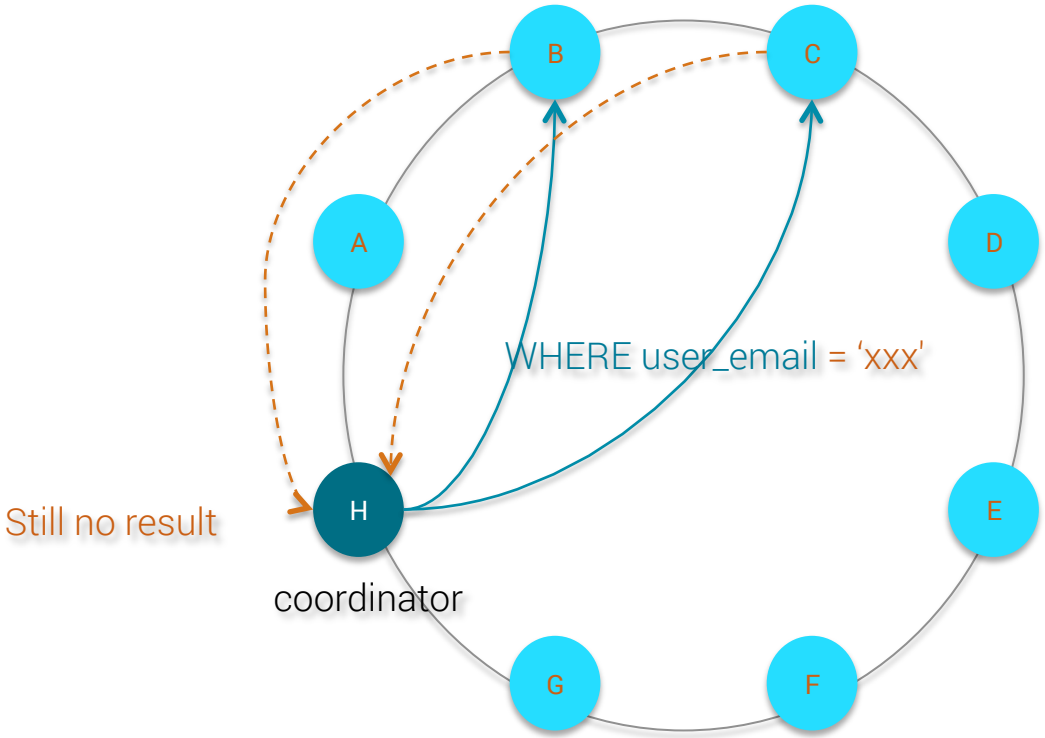
SELECT *
FROM …
WHERE …
LIMIT 1000



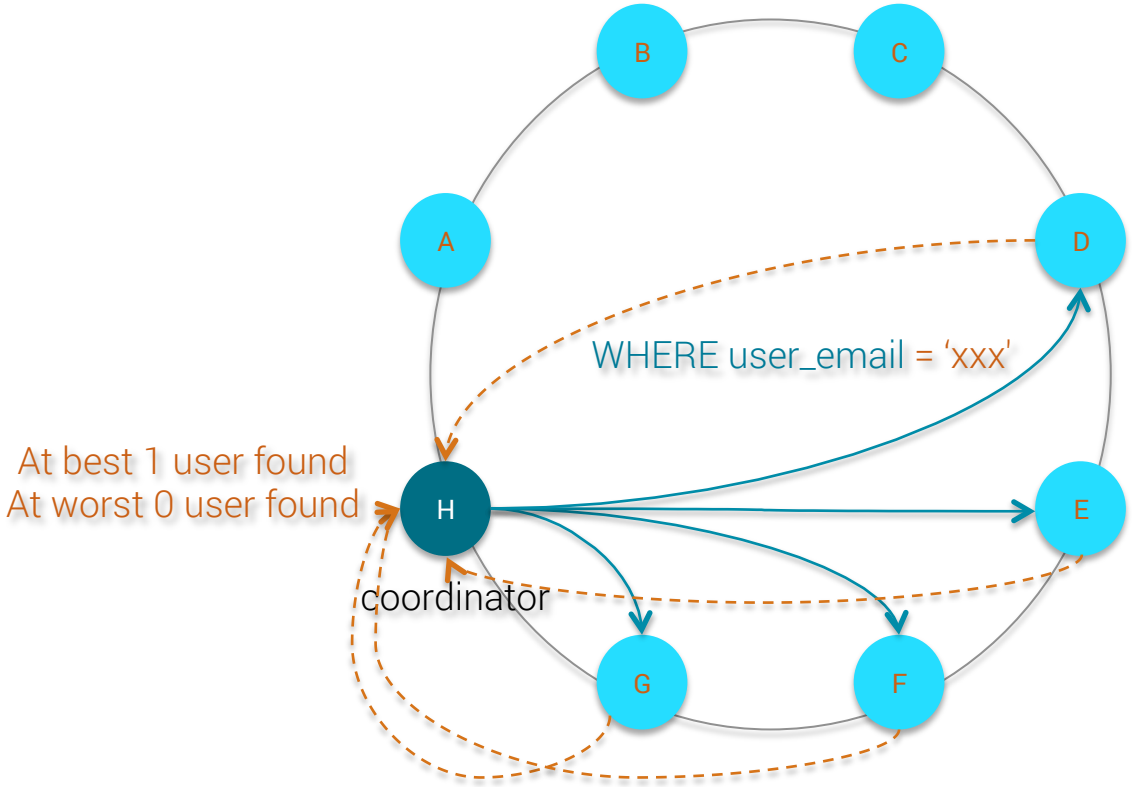
Caveat 2: 1-to-1 index (*user_email*)



Caveat 2: 1-to-1 index (*user_email*)



Caveat 2: 1-to-1 index (*user_email*)



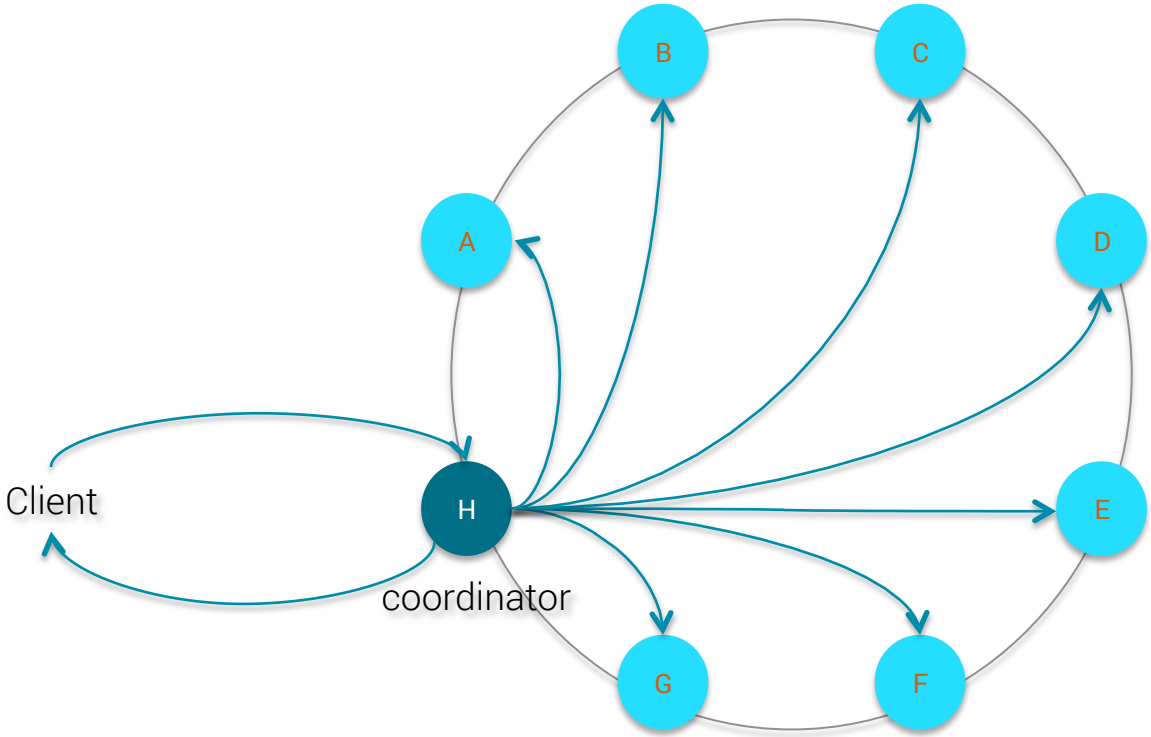
Caveat 2 solution: materialized views

For 1-to-1 index/relationship, use materialized views instead

```
CREATE MATERIALIZED VIEW user_by_email AS
SELECT * FROM users
WHERE user_id IS NOT NULL and user_email IS NOT NULL
PRIMARY KEY (user_email, user_id)
```

But range queries (<, >, ≤, ≥) not possible …

Caveat 3: fetch all rows for analytics use-case



Caveat 3 solution: use co-located Spark

Local index query

SASI local read/write path

Local write path

Index files are built

- on memtable flush
- on compaction flush

To avoid OOM, index files are split into chunk of

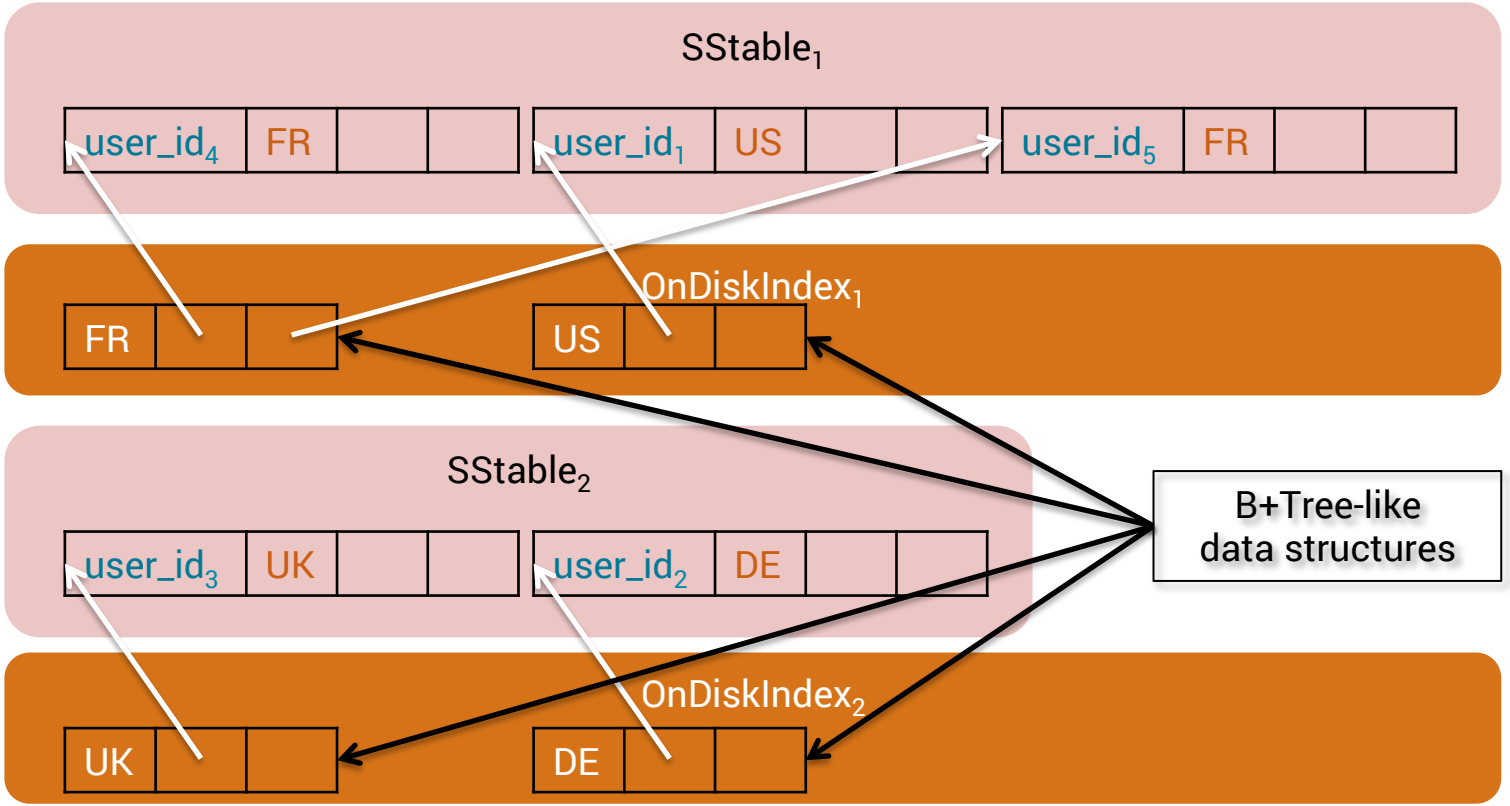
- 1Gb for memtable flush
- *max_compaction_flush_memory_in_mb* for compaction flush

Local write path data structures

Index mode, data type	Data structure	Usage
PREFIX, text	Guava *ConcurrentRadixTree*	name LIKE 'John%'
CONTAINS, text	Guava *ConcurrentSuffixTree*	name LIKE '%John%' name LIKE '%ny'
PREFIX, other	JDK *ConcurrentSkipListSet*	age = 20 age >= 20 AND age <= 30
SPARSE, other	JDK *ConcurrentSkipListSet*	age = 20 age >= 20 AND age <= 30

suitable for 1-to-N index with N ≤ 5

OnDiskIndex files

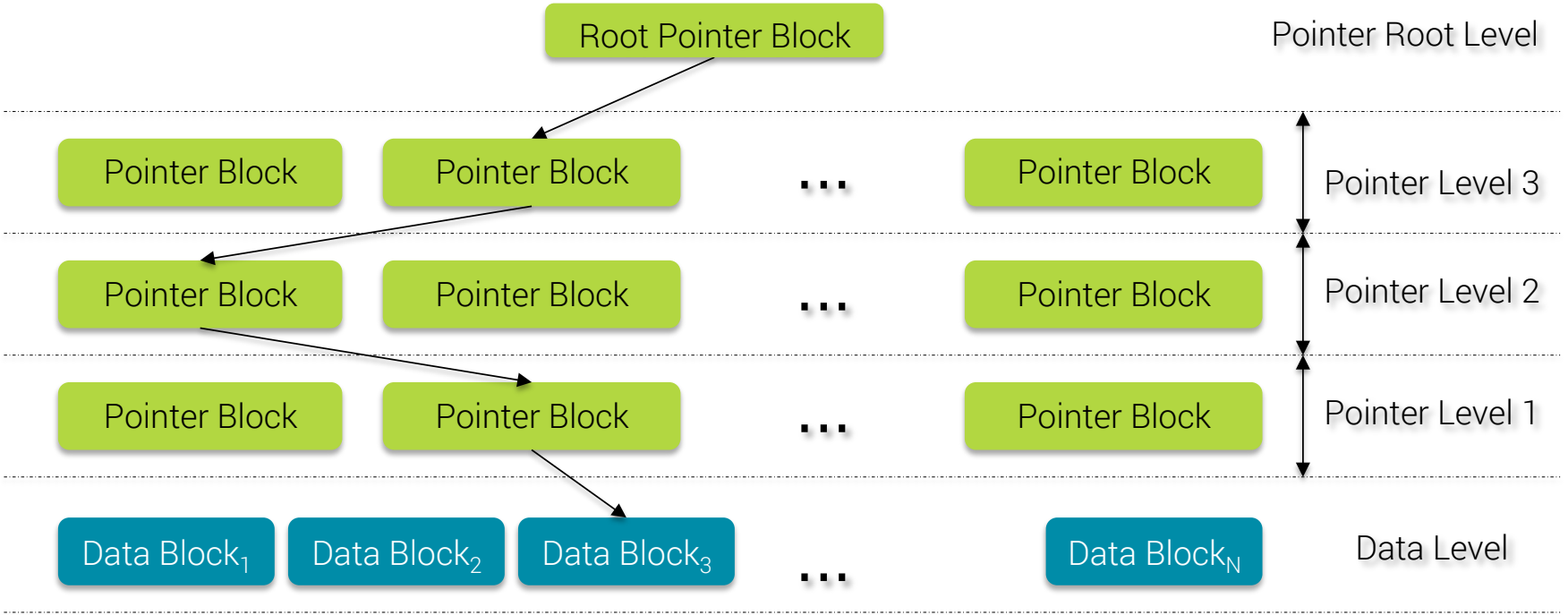


Local read path

- first, optimize query using Query Planer (see later)
- then load chunks (4k) of index files from disk into memory
- perform binary search to find the indexed value(s)
- retrieve the corresponding **partition keys** and push them into the **Partition Key Cache**

→ Yes, **currently** SASI only keep partition key(s) so on wide partition it's not very optimized ...

Binary search using OnDiskIndex files



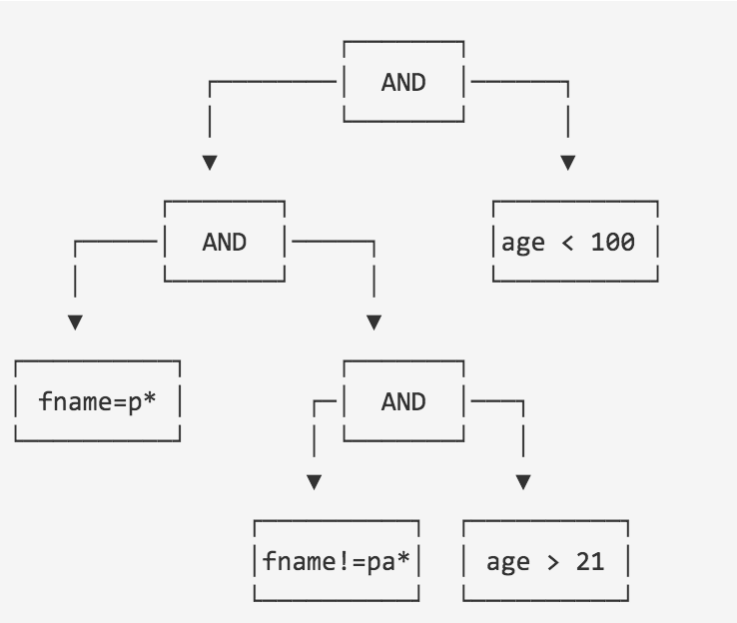
Query Planner

Query planner

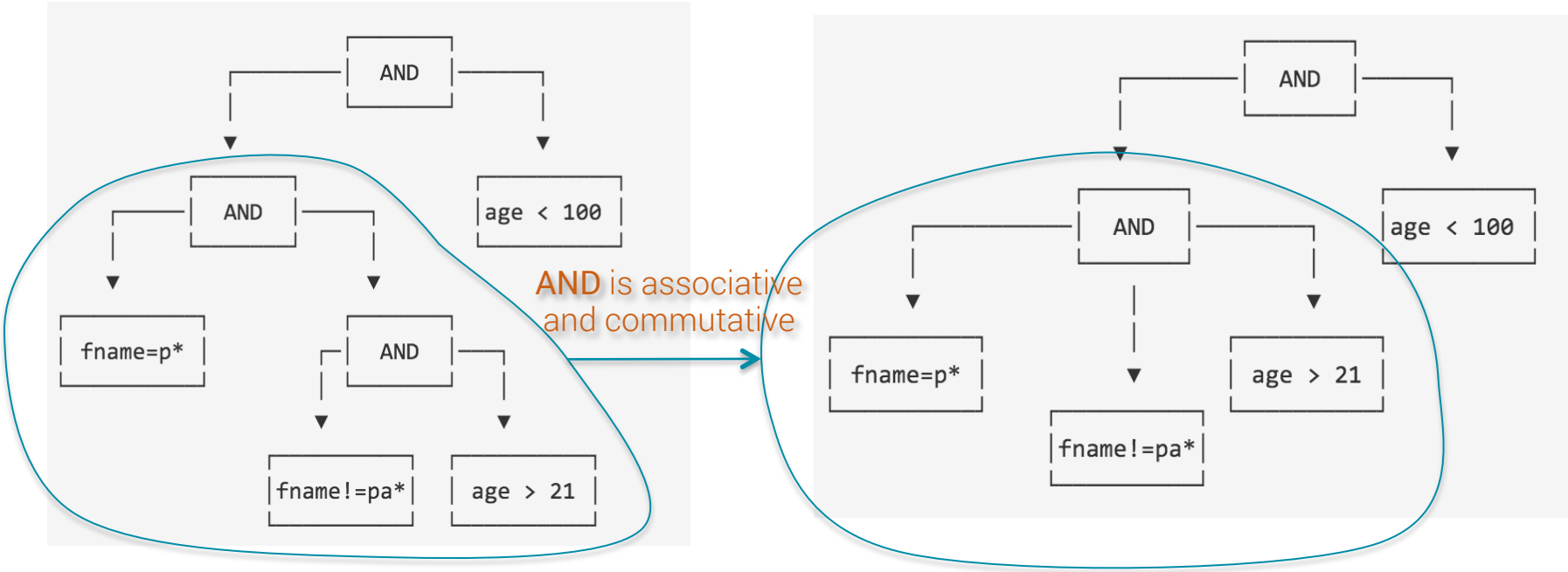
- build predicates tree
- predicates push-down & re-ordering
- predicate fusions for != operator

Query optimization example

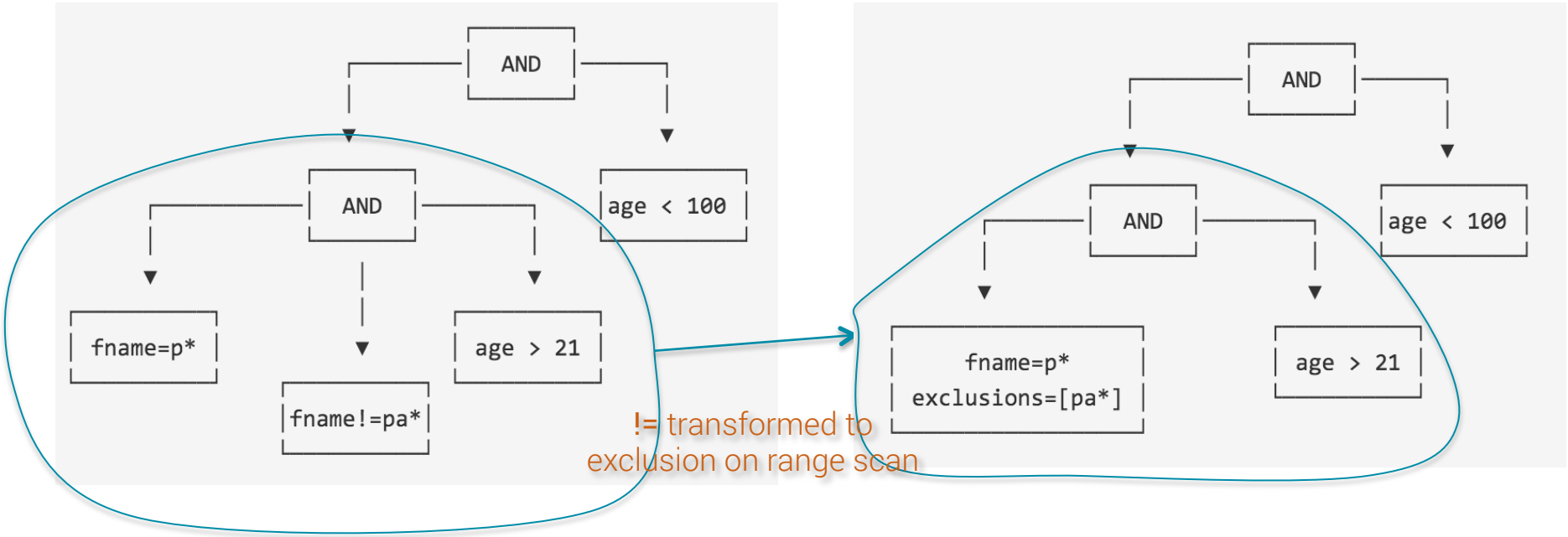
WHERE age < 100 AND fname LIKE 'p%' AND fname != 'pa%' AND age > 21



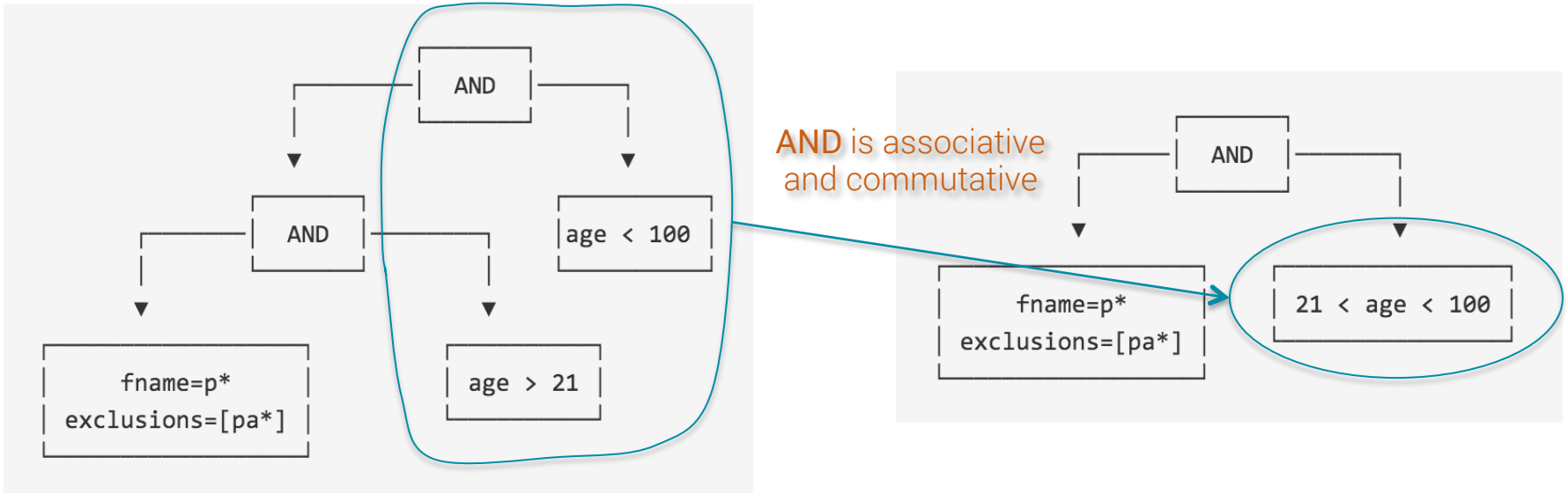
Query optimization example



Query optimization example



Query optimization example



Some benchmarks

Hardware specs

13 bare-metal machines

- 6 CPU HT (12 vcores)
- 64Gb RAM
- 4 SSDs in RAID0 for a total of 1.5Tb

Data set

- 13 billions of rows
- 1 numerical index with 36 distinct values
- 2 text index with 7 distinct values
- 1 text index with 3 distinct values

Benchmark results

Full table scan using co-located Spark (no LIMIT)

Predicate count	Fetched rows	Query time in sec
1	36 109 986	609
2	2 781 492	330
3	1 044 547	372
4	360 334	116

Benchmark results

Full table scan using co-located Spark (no LIMIT)

Predicate count	Fetched rows	Query time in sec
1	36 109 986	609
2	2 781 492	330
3	1 044 547	372
4	360 334	116

Benchmark results

Beware of disk space usage for full text search !!!

Table *albums* with ≈ 110 000 records, 6.8Mb data size

Index Name	Index Mode	Analyzer	Index Size	Index Size/SSTable Size Ratio
albums_country_idx	**PREFIX**	**`NonTokenizingAnalyzer`**	2Mb	0.29
albums_year_idx	**PREFIX**	N/A	2.3Mb	0.34
albums_artist_idx	**CONTAINS**	**`NonTokenizingAnalyzer`**	**30Mb**	**4.41**
albums_title_idx	**CONTAINS**	**`StandardAnalyzer`**	**41Mb**	**6.03**

Take Away

SASI vs search engines

SASI vs Solr/ElasticSearch ?

- Cassandra is not a search engine !!! (database = durability)
- always slower because 2 passes (SASI index read + original Cassandra data)
- no scoring
- no ordering (~~ORDER BY~~)
- no grouping (~~GROUP BY~~) → Apache Spark for analytics

If you don't need the above features, SASI is for you!

SASI sweet spots

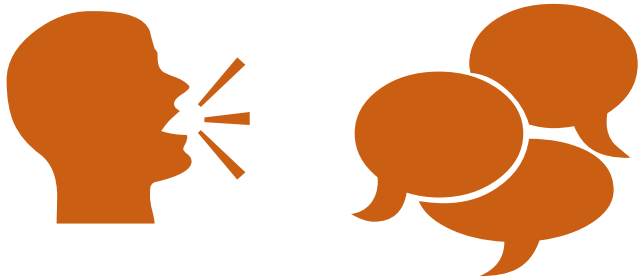
SASI is a relevant choice if

- you need **multi criteria search** and you don't need ordering/grouping/scoring
- you mostly need **100 to 10000** of rows for your search queries
- you **always know the partition keys** of the rows to be searched for (this one applies to native secondary index too)
- you want to index **static columns** (**SASI** has no penalty since it indexes the whole partition)

SASI blind spots

SASI is a poor choice if

- you have **very wide partitions** to index, SASI only indexes the partition offset (but it will change with **CASSANDRA-11990** merged to trunk)
- you have **strong SLA on search latency**, for example few millisecs requirement
- **ordering of the search results** is important for you



Q & A

Thank You

@doanduyhai

duy_hai.doan@datastax.com

https://academy.datastax.com/

CASSANDRA SUMMIT 2016