# Performance Enhancement of Apache Spark's Broadcast Hash Joins


## Abstract

Apache Spark's Broadcast Hash Join (BHJ) is widely used for efficiently joining large datasets with smaller dimension tables by broadcasting the smaller dataset to all executors. Despite its effectiveness, BHJ can incur unnecessary scan overhead when the stream side reads large volumes of data that are not relevant to the join keys. This paper presents a strategy to enhance BHJ performance by leveraging broadcast data as a SortedSet filter predicate at the scan level. The proposed approach utilizes the column statistics (e.g., Parquet min/max values) to prune irrelevant data blocks before reading, significantly improving I/O efficiency and overall query execution time.


## 1. Introduction

Apache Spark is a leading distributed computing engine used for large-scale data processing. Among its join strategies, Broadcast Hash Join offers substantial advantages when one of the datasets is small enough to be broadcast to all executors. However, as data volumes and schema complexity grow, the cost of evaluation of nested joins increase.

This paper proposes a lightweight, backward-compatible optimization to enhance BHJ performance by pushing broadcasted data down to the scan layer as a SortedSet-based filter, enabling early data pruning during the stream-side read.

## 2. Background on Broadcast Hash Joins

In Spark's Broadcast Hash Join:
- The build side (small dataset) is broadcast to all executors.
- The stream side (large dataset) is scanned and joined against the broadcasted data.
- The broadcasted data is available before the stream-side iterator is opened.

While this design minimizes shuffle and network overhead, the stream side still scans potentially large data volumes — even when many blocks are irrelevant to the join. Existing implementation rely on partition-level statistics and filter pushdown only if the joining stream side column is a partitioning column ( The Dynamic Partition Pruning concept).

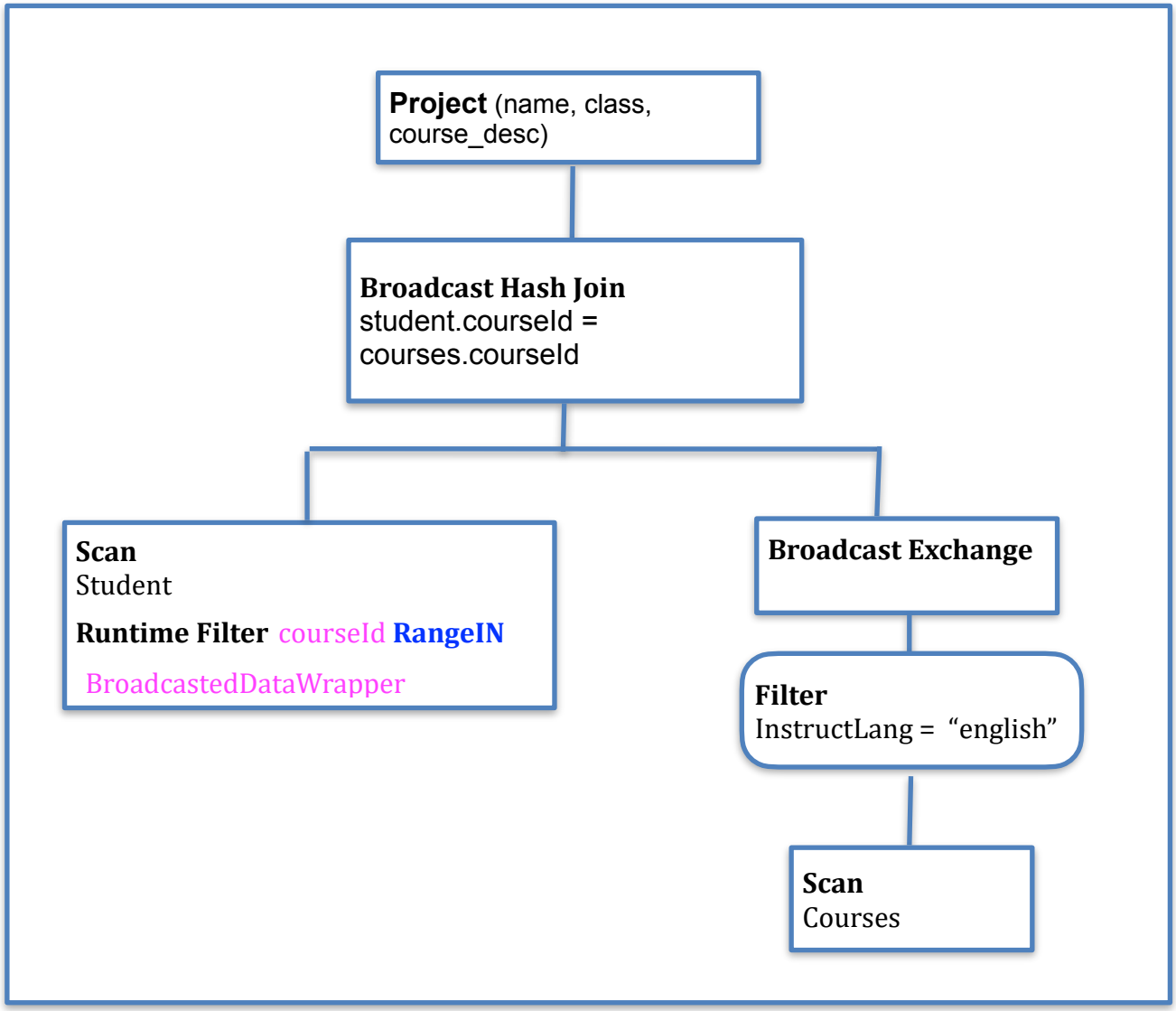
## 3. Proposed Optimization Approach using RangeIn filter

The core idea is to use the broadcasted build-side dataset as a SortedSet contained in a new type of Filter called RangeIn, if the join key implements the Comparable interface (such as Long, String, Date, or Integer). The key difference between RangeIn filter and the usual In filter is that former contains the Comparable values in a SortedSet. . The advantages are that

it works even if the stream side joining column is a regular column ( i.e not partitioned) and there is no extra overhead of obtaining build data as its already available, eliminating the need for dynamic pruning query on the build side.

Example Query:

Select name, class, course_desc from **students** join **courses** on **students.courseId = courses.courseId** where courses.instructLang = “english”



### 3.1 RangeIn Filter Pushdown via SortedSet

- Convert the broadcasted join keys into a SortedSet, encapsulated by RangeIn Filter
- Pass this set to the stream-side scan before the iterator opens.
- During data reading, compare each block's column-level min/max statistics against the broadcasted key range.
- Skip data blocks where its guaranteed that for the range spanning min/max of the block, have empty subset in the sorted set.

This approach leverages metadata available in columnar formats (e.g., Parquet, ORC) without requiring data deserialization.

### 3.2 Columnar vs Row-Level Filtering

When data is read as columnar batches, per record-level filtering cannot occur at the scan level, as the data is in a columnar batch.. However, row level filtering can still be applied during Column-to-Row transformation, in case of nested BHJs, ensuring the optimization still provides measurable benefit even with columnar reads.

## 4. Implementation Considerations

### 4.1 Simple Join Scenarios

In single-join or non-adaptive plans, implementation is straightforward: the broadcasted dataset is materialized early, and the scan operator can easily access the SortedSet before opening the iterator.

### 4.2 Nested Broadcast Joins

In realistic queries, multiple nested BHJs are common. In such cases, its possible that multiple such filters may get pushed to same or different leaf scans. Care needs to be taken that a stream side scan does not get opened, until it has got all the relevant Broadcasted filters pushed. Then there are certain operators like Aggregate, windows or Joins of type Outer, which if existing below the Broadcast Hash Join, cannot allow push down of the Broadcasted Keys filter.

## 5. Integration with Adaptive Query Execution (AQE)

Adaptive Query Execution (AQE) introduces significant complexity: stages are created for Exchange operators and materialized asynchronously, and plans may be re-optimized after each stage is materialized. Identical exchanges may be reused across stages.

Because the decision to push down broadcast data cannot be made during the optimization phase (when join strategies are undecided), the system must defer pushdown decisions until planning phase. But even in the planning phase, the materialized Broadcast Data is not available, as it is materialized only during physical plan execution. Moreover, pushing down broadcast data affects exchange reuse: previously identical plans may diverge if one includes broadcast filters. Hence, the AQE planner must detect and manage these divergences gracefully to prevent incorrect plan reuse.

The Broadcasted Data needs a proxy to represent itself as a filter to the underlying scan, till the actual broadcasted data is materialized.

In case of Adaptive Query Execution, an exchange is represented as a stage. Each stage is fetched asynchronously  starting from Leaves to Root. Care needs to be taken to delay fetching of a stage, till the leaf scans of that Exchange,  have their respective materialized broadcast filters pushed .

## 6. Challenges and Mitigation Strategies

| Challenge | Impact | Mitigation Strategy |
|---|---|---|
| Stage should not be fetched, till relevant Broadcast Filters have been pushed to the leaf scans. | Premature opening of scan iterator would loose optimization. | Asynch Stage fetching should be coordinated such that fetch is started after pushdown of all filters. |
| Exchange reuse with differing filters | Incorrect reuse of cached plans | Track filter lineage as part of exchange plan signature |
| Columnar read limitations | Filtering granularity reduced | Apply filter during Column-to-Row transition in case of nested BHJs. |

## 7. Performance Implications

The proposed optimization reduces I/O and CPU overhead in the following ways:
- Block-level pruning: Reduces unnecessary reads for irrelevant data pages.
- Reduced deserialization cost: Fewer records pass through the pipeline before join filtering.

Preliminary internal benchmarks show that on TPC-DS-like workloads with large fact tables and selective dimension joins, show end-to-end query time improvements of 36% or can be achieved without modifying the user query or dataset schema.

## 8. Conclusion and Future Work

This paper presents a practical approach to enhancing Apache Spark's Broadcast Hash Join by pushing broadcasted join keys as a SortedSet-based filter to the stream-side scan. By leveraging metadata-level pruning and column statistics, the method offers tangible performance improvements with minimal implementation overhead.

Future work will focus on extending this optimization to multi column-joins. In case of multi column join, separate Broadcast Filters are pushed which looses the pairing information, as a result the pruning will be sub -optimal.

Author: Asif Hussain Shahid

Date: October 2025

Keywords: Apache Spark, Broadcast Hash Join, Performance Optimization, Filter Pushdown, Adaptive Query Execution