Motivation

The current ***scan.partition.column*** implementation splits data by equal numeric intervals, which breaks down when data is skewed or when no suitable numeric column exists. This causes two practical problems:

- For environments where CDC adoption is not feasible, JDBC full-scan import is the primary alternative. Without reliable partitioning, large-scale ingestion is prone to skew-induced instability and poor throughput.
- Even when CDC is adopted, the initial snapshot phase requires a full-scan read of the source table over JDBC. Skewed partitioning at this stage risks task failures during the most critical phase of the pipeline.
- When equal-interval partitioning is insufficient due to data characteristics or business requirements, users need a way to apply a more effective distribution strategy based on their knowledge of the source data.

This FLIP proposes two extensions to address these:

- **Physical ID partitioning** for databases that expose a physical row identifier (e.g. Oracle, PostgreSQL), enabling skew-free parallel reads with no user-defined partition column required.
- **Boundary query** allows users to define partition boundaries based on actual data distribution, enabling balanced parallel reads for any column type.

Basic Idea

Two complementary approaches are introduced:

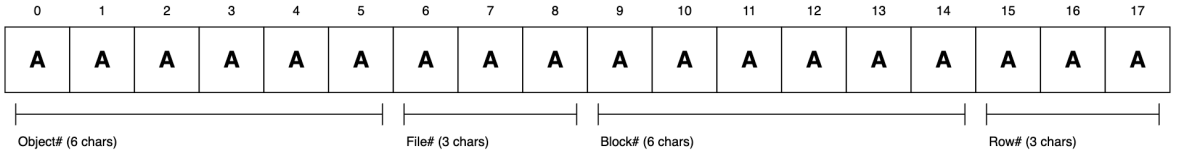
Physical ID Partitioning

Databases expose a physical row identifier that encodes the storage block location of each row. Since data is physically distributed across blocks, partitioning by block location naturally produces balanced splits without any knowledge of the data distribution. This feature is limited to databases that natively support such identifiers.

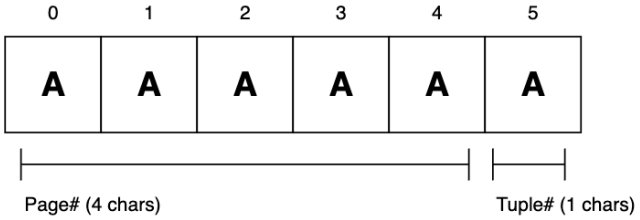
Database	**Physical Identifier**	**Supported**
Oracle	ROWID	O
PostgreSQL	ctid	O
MySQL	-	X

SQL Server	-	X
DB2	-	X
Others	-	X

Oracle (ROWID)



PostgreSQL (ctid)



Boundary Query

For cases where the user has domain knowledge of the data distribution, a ***scan.partition.boundary-query*** option allows them to provide a SQL query that returns ordered boundary values. The connector uses these values to generate range predicates, giving users full control over partition balance regardless of column type.

This is conceptually close to ***Apache Sqoop***'s ***--boundary-query*** option (see Sqoop User Guide). The primary difference is that string-type columns are supported natively: where Sqoop required converting strings to numeric values (e.g., via ***ascii()***) to use them as split boundaries, this option accepts string values directly.

scan.partition.num is required when ***scan.partition.boundary-query*** is set. The first ***partition.num - 1*** boundary values are used as split points, and the last partition absorbs all remaining data beyond that point.

Public Interfaces

New option: scan.partition.use-physical-id (DB-specific)

Defined per dialect for databases that natively expose a physical row identifier. Enables skew-free partitioning based on physical storage order without requiring a user-defined partition column. Partition bounds are derived automatically from DB-native metadata. ***scan.partition.num*** is required. Cannot be combined with ***scan.partition.column***, ***scan.partition.lower-bound***, or ***scan.partition.upper-bound***.

Added to each supporting dialect's options class (e.g. ***OracleConnectorOptions***, ***PostgresConnectorOptions***):

```
public static final ConfigOption<Boolean> SCAN_PARTITION_USE_PHYSICAL_ID =
        ConfigOptions.key("scan.partition.use-physical-id")
                .booleanType()
                .defaultValue(false)
                .withDescription(
                        "Enable partitioning based on the physical row identifier provided by the
database. "
                                + "Partition bounds are derived automatically from DB-native metadata. "
                                + "Requires 'scan.partition.num'. "
                                + "Cannot be combined with 'scan.partition.column', "
                                + "'scan.partition.lower-bound', or 'scan.partition.upper-bound'.");
```

Extended: scan.partition.column

Currently, ***scan.partition.column*** accepts only numeric, date, and timestamp column types. The partition bounds are ***Long***-typed (***scan.partition.lower-bound***, ***scan.partition.upper-bound***), which makes string columns structurally unsupported.

This FLIP extends ***scan.partition.column*** to accept string-type columns, but only when ***scan.partition.boundary-query*** is also set. Without ***boundary-query***, specifying a string-type column is a validation error. This dependency is enforced at configuration validation time, not at runtime.

scan.partition.column type	**scan.partition.boundary-query**	**Result**
Numeric / Date / Timestamp	Not set	Existing behavior (lower-bound / upper-bound required)
Numeric / Date / Timestamp	Set	boundary-query used for split points
String	Not set	Validation error
String	Set	Supported

New option: scan.partition.boundary-query

A SQL query returning ordered boundary values for the column specified in ***scan.partition.column***. Allows users to define custom partition boundaries based on the actual data distribution, enabling balanced parallel reads regardless of column type. ***scan.partition.num*** is required. Cannot be combined with ***scan.partition.lower-bound*** or ***scan.partition.upper-bound***.

Up to ***partition.num - 1*** boundary values are used as split points. Fewer values reduce the actual partition count; more values are a validation error. The last partition absorbs all remaining data beyond the last split point.

Added to ***JdbcConnectorOptions***:

```
public static final ConfigOption<String> SCAN_PARTITION_BOUNDARY_QUERY =
        ConfigOptions.key("scan.partition.boundary-query")
                .stringType()
                .noDefaultValue()
                .withDescription(
                        "A SQL query returning ordered boundary values for the column "
                        + "specified in 'scan.partition.column'. "
                        + "Requires 'scan.partition.num'. "
                        + "The query must return a single column whose type is comparable with the
partition column. "
                        + "NULL boundary values are not allowed. "
                        + "Up to 'scan.partition.num - 1' boundary values are used as split points; "
                        + "fewer values reduce the partition count, more values are a validation error. "
                        + "Cannot be combined with 'scan.partition.lower-bound' or
'scan.partition.upper-bound'.");
```

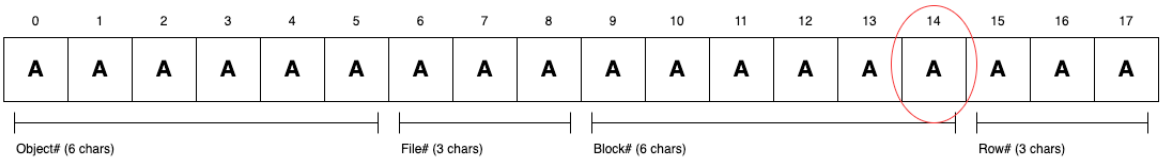
Proposed Changes

Physical ID Partitioning

Databases such as Oracle (***ROWID***), PostgreSQL (***ctid***) expose a physical row identifier that reflects storage block location. Because data is distributed uniformly across storage blocks, partitioning by this value produces naturally balanced splits without any knowledge of the data distribution.

Each supporting dialect contains the physical identifier expression and the bounds query logic internally. The connector uses these to generate range predicates at job startup. This feature is defined at the dialect level and is only available for databases that natively support it.

Oracle — ascii(substr(rowid, -4, 1)) (Oracle ROWID Pseudocolumn)



Oracle ROWID is an 18-character Base64-encoded string with four fields: Object# (6), File# (3), Block# (6), Row# (3). ***substr(rowid, -4, 1)*** extracts the last character of the Block# field (position 15). ***ascii()*** converts it to a numeric value usable in BETWEEN predicates.

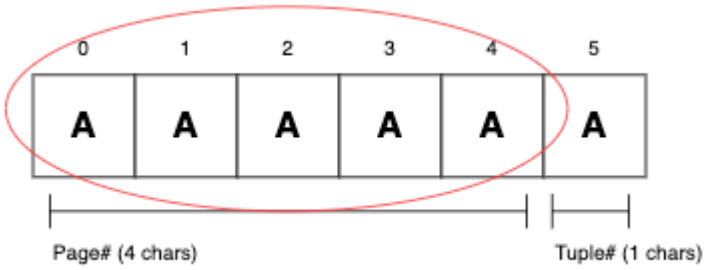
Object# and File# are identical across most rows in the same table. Row# represents order within a block, which causes skew. The last character of Block# cycles as blocks fill up, distributing rows across the numeric range naturally.

```
-- bounds via full table scan
SELECT MIN(ascii(substr(rowid,-4,1))), MAX(ascii(substr(rowid,-4,1)))
FROM your_table;

-- bounds via data dictionary (no full table scan)
-- Base64 character range is fixed, so fixed bounds can be used
SELECT ascii('A') AS lower_bound, ascii('z') AS upper_bound FROM dual;

-- partition predicate
SELECT * FROM your_table
WHERE ascii(substr(rowid,-4,1)) BETWEEN 65 AND 78
```

PostgreSQL — (ctid::text::point)[0]::bigint (PostgreSQL System Columns)



PostgreSQL ***ctid*** is a (**page, tuple**) pair representing the physical row location. The page number is extracted and cast to bigint. Page numbers increase monotonically as data is written, so equal-range splits on page number produce naturally balanced partitions. The upper bound is read from **pg_class.relpages**, avoiding a full table scan.

```
-- bounds via catalog (no full table scan)
```

```
SELECT 0, relpages FROM pg_class WHERE relname = 'your_table';

-- partition predicate
SELECT * FROM your_table
WHERE (ctid::text::point)[0]::bigint BETWEEN 0 AND 249
```

Correctness Guarantees and Limitations

This mode is suitable for static tables, or tables with no concurrent changes during the scan. Concurrent structural or DML operations can ca
use rows to be missed or read twice across split boundaries depending on the database.

Physical row identifiers are managed at the storage layer, outside MVCC. Each split uses an independent connection with no global transaction, so cross-split snapshot consistency is not provided regardless of partitioning mode. Stability of the identifier itself differs by database:

- **Oracle ROWID**: Stable under DML (INSERT/UPDATE/DELETE). Only structural operations (***ALTER TABLE MOVE***, row movement, EXPORT/IMPORT) reassign ROWID values — these must not run concurrently with a scan.
- **PostgreSQL ctid**: Changes on every ***UPDATE*** (new tuple written at a new ctid) and on ***VACUUM FULL*** / ***CLUSTER***. Only safe when no concurrent DML runs during the scan — use for read-only or append-only tables.

Boundary Query

The existing equal-interval partitioning produces skew when data is not uniformly distributed. The ***scan.partition.boundary-query*** option allows users who understand their data distribution to define partition boundaries explicitly via a SQL query. The connector uses the query result to generate range predicates, enabling balanced parallel reads for any column type. This is conceptually close to ***Apache Sqoop***'s ***--boundary-query*** option (see Sqoop User Guide). The primary difference is native support for string-type partition columns: where Sqoop required converting string values to numeric (e.g., via ***ascii()***) to use them as split boundaries, this option accepts string values directly in the boundary-query result.

Boundary-query result requirements:

The boundary query must satisfy the following constraints, enforced at job startup:

- **Single column** : The query must return exactly one column. Multi-column results are a validation error.
- **Type compatibility** : The returned column's type must be comparable with the type of `scan.partition.column`. Type mismatches are a validation error.
- **No NULL boundary values** : If any returned boundary value is NULL, the job is rejected. NULL split points produce meaningless predicates (`col < NULL` always evaluates to NULL) and cannot be used to define partition ranges. Note that this applies to the boundary values themselves — NULL values in the data column are handled separately (routed to Partition 1 via `col IS NULL`).

Column type validation: When ***scan.partition.column*** refers to a string-type column, ***scan.partition.boundary-query*** must be present. The connector enforces this dependency at configuration validation time and rejects the job before execution if the constraint is violated.

Difference from ***lower-bound*** / ***upper-bound***:

lower-bound and ***upper-bound*** divide the given range into equal intervals mechanically. This works only for numeric columns and assumes uniform data distribution within the range.

```
lower=1, upper=10000, num=3
→ 1~3333 | 3334~6666 | 6667~10000  (equal intervals, regardless of actual distribution)

Actual data: 1~100 (10,000 rows), 101~1000 (100 rows), 1001~10000 (10,000 rows)
→ Partition 1 overloaded, Partition 2 nearly empty  ← skew
```

boundary-query lets users specify split points based on where the data actually is, for any column type:

```
boundary-query → ["100", "1001"]
→ col < 100 | 100 <= col < 1001 | col >= 1001  (distribution-aware split)

→ balanced partitions
```

The key distinction: ***lower/upper-bound*** requires knowing the data range; ***boundary-query*** requires knowing where the data is concentrated.

Predicate contract:

Given ***N = scan.partition.num*** and boundary values ***v1, v2, ..., v(N-1)*** (ordered ascending):

Partition	Predicate
1 (first)	col < v1 OR col IS NULL
2 ~ N-1 (middle)	v(i-1) <= col < vi
N (last)	col >= v(N-1)

NULL rows are absorbed into the first partition by the explicit ***col IS NULL*** clause. This is necessary because SQL NULL comparisons (***col < v1***) always return ***NULL*** (not ***TRUE***), which means NULL rows would be silently dropped without the explicit clause. This matches the behavior of the existing ***lower-bound / upper-bound*** mode.

Middle partitions are left-inclusive and right-exclusive, which ensures no row is emitted by two partitions simultaneously when boundary values are distinct.

ORDER BY push-down:

The connector wraps the user-provided scan.partition.boundary-query in a subquery and appends ORDER BY 1 before execution:

SELECT * FROM (<user-boundary-query>) AS boundary ORDER BY 1

This ensures the split points are always retrieved in ascending order, even if the user's query does not include ORDER BY. The database can use an existing index on the partition column to satisfy the sort without an additional sort step.

Partition count behavior:

scan.partition.num = N means the query is expected to return at most **N-1** boundary values. Two outcomes are possible:

- **Fewer than N-1 values returned**: The actual partition count is reduced to **(returned values) + 1**. This is allowed. For example, if **scan.partition.num = 5** but the query returns 2 values, 3 partitions are created.
- **More than N-1 values returned**: This is a validation error. **scan.partition.num** serves as a resource cap (controls parallelism and downstream pressure). Exceeding it would silently create more partitions than the user intended. The job is rejected at startup with a descriptive error message.

Example ***partition.num = 3***:

```
boundary-query result: ["electronics", "fashion"]   ← exactly N-1 values
→ 3 partitions:
  Partition 1: col < 'electronics' OR col IS NULL
  Partition 2: 'electronics' <= col < 'fashion'
  Partition 3: col >= 'fashion'

boundary-query result: ["electronics"]              ← fewer than N-1 values
→ 2 partitions (reduced):
  Partition 1: col < 'electronics' OR col IS NULL
  Partition 2: col >= 'electronics'

boundary-query result: ["a", "b", "c"]              ← more than N-1 values → error
→ Validation error: boundary-query returned 3 values but scan.partition.num=3 allows at
most 2.
```

Option Validation Matrix

The three partitioning modes (Range, Boundary-query, Physical-id) have mutually exclusive option sets. The table below summarizes which options are required, optional, or disallowed for each mode.

Option	Range mode	Boundary-query mode	Physical-id mode
scan.partition.column	Required	Required	Not allowed
scan.partition.lower-bound	Required	Not allowed	Not allowed

scan.partition.upper-bound	Required	Not allowed	Not allowed
scan.partition.num	Required	Required	Required
scan.partition.boundary-query	Not allowed	Required	Not allowed
scan.partition.use-physical-id	Not allowed	Not allowed	Required

Notes:

- ***scan.partition.use-physical-id*** is defined per dialect (e.g. ***OracleConnectorOptions***, ***PostgresConnectorOptions***) and is only available for databases that natively support physical row identifiers.
- String-type ***scan.partition.column*** is only valid in boundary-query mode. Using a string column in Range mode is a validation error.
- All three modes require ***scan.partition.num*** to control parallelism.

Compatibility

Existing behavior is unchanged. Users who currently use ***scan.partition.column*** with numeric or date columns and explicit bounds continue to work without any modification.

Test Plan

- Validation: all valid and invalid option combinations for new options
- Validation: string-type ***scan.partition.column*** without ***scan.partition.boundary-query*** is rejected at configuration time
- Validation: string-type ***scan.partition.column*** with ***scan.partition.boundary-query*** is accepted
- Validation: ***scan.partition.boundary-query*** without ***scan.partition.num*** is rejected at configuration time
- Validation: boundary-query returning more than `scan.partition.num - 1` values is rejected
- Validation: boundary-query returning NULL values is rejected
- Validation: boundary-query returning multiple columns is rejected
- Validation: boundary-query returning a type incompatible with `scan.partition.column` is rejected
- Integration: Oracle, PostgreSQL full-scan with ***scan.partition.use-physical-id***, verify row count and partition balance
- Integration: partitioning via ***scan.partition.boundary-query*** with ***scan.partition.num***, verify last partition absorbs remaining data and no data loss
- Regression: existing numeric partition behavior unchanged

Rejected Alternatives

DB2 Physical ID Partitioning via ***RID()*** DB2 exposes a ***RID()*** function that returns the physical row identifier as a BIGINT encoding the page number and slot, which is conceptually equivalent to Oracle's ROWID and PostgreSQL's ctid. However, DB2's query optimizer does not treat ***WHERE RID(t) BETWEEN :low AND :high*** as a physical block range scan. Instead, it performs a full table scan with a post-filter evaluation for each split, meaning N parallel splits result in N full table scans rather than one scan distributed across N partitions. This negates the performance benefit of physical ID partitioning. Rejected on the grounds that range scan performance cannot be guaranteed.

Automatic lexicographic range splitting dividing the character space between min and max string values produces severe skew when the data distribution does not match the character space. Rejected in favor of user-defined boundary query.

Hash modulo partitioning ***WHERE MOD(HASH(col), N) = ?*** was considered as a built-in partitioning strategy. Rejected because boundary-query is a more general approach that subsumes it. Users who need hash-based partitioning can express it within a boundary-query.