A Subset-Aware Scheduling Method for ELT Pipelines in Apache Hudi and Its Application to Equity Market Data

2230014 Yo Hasebe (Inoguchi Laboratory on Massively Parallel Systems)

[**Background and purpose**] In a lakehouse, near-real-time analytics depends on how quickly continuously arriving stream data is reflected in downstream tables. Many systems support incremental queries that read only the records appended or updated since the previous processing time, and this is a natural building block for frequent ELT (Extract–Load–Transform) executions.

However, incremental queries are not always efficient. When incremental records are dispersed across many data files, the engine still has to touch many files to discover and read those records. This dispersed-file regime tends to occur when updates are spread across many primary keys with high cardinality, and it is amplified when frequent commits create many small files. Even if each file contains only a small number of new records, per-file overhead (metadata access and planning) accumulates, increasing ELT execution time and prolonging staleness in downstream tables.

Freshness-oriented scheduling aims to mitigate this issue by prioritizing jobs that reduce staleness the most per unit cost. A representative approach, Max-Benefit, evaluates each job using the expected staleness reduction achieved by completing the job divided by its expected execution cost, and then greedily prioritizes jobs by this value. Yet existing formulations usually assume that a job processes *all* incremental input files available at that time. In the dispersed-file regime, processing all incremental files can have low benefit per time because many low-yield files contribute little to freshness relative to their overhead. This thesis addresses this gap by introducing file-level subset selection as a first-class part of scheduling for Apache Hudi-based ELT pipelines.

[**Experimental results**] We propose *Partitioned-Max-Benefit*, a scheduler that extends job evaluation from "process all incremental files" to "process a selected subset of incremental files when beneficial." The scheduler runs as a resident process and launches Spark jobs with the chosen file list as an input parameter. For each job j, it first identifies the set of candidate incremental files F_j using Apache Hudi timeline metadata under `.hoodie`. From each `.commit` file, it extracts file paths and file sizes through `partitionToWriteStats` (i.e., `WriteStat`) so that it can reason about file-level costs without scanning full data.

To estimate the "recency" of each file without reading its contents, the scheduler reads the Parquet footer and uses column statistics for the *latest-arrival-time* column. Specifically, for each file $f \in F_j$, it obtains the maximum value $t_j^{\max}(f)$ from Parquet statistics. The scheduler also maintains the reflected latest-arrival time t_j^{done}, which represents how far the destination table has already incorporated arrivals for job j.

Given these per-file estimates, Partitioned-Max-Benefit enumerates candidate subsets using a prefix-based restriction. Files in F_j are sorted by $t_j^{\max}(f)$ in ascending order, and candidates are the prefix subsets $C_{j,k} = \{f(1), \ldots, f(k)\}$ for $k = 1, \ldots, |F_j|$. This keeps the search space tractable while preserving consistent progress semantics: after executing a subset, the scheduler updates t_j^{done} based on the chosen subset and completion.

For each candidate subset C for job j, the scheduler estimates (a) the staleness reduction $G_j(C)$ and (b) the execution time $E_j(C)$, and computes marginal-benefit $G_j(C)/E_j(C)$. The coefficients in the time estimator are obtained by linear regression using observed execution times and input characteristics. The scheduler selects $C_j^* = \arg\max_C G_j(C)/E_j(C)$ and then prioritizes jobs greedily using these maximized values, similarly to Max-Benefit but with subset-aware evaluation. We implemented the scheduler as a resident Python process that polls metadata, computes decisions, and submits Spark jobs as child processes (e.g., via `spark-submit`) while monitoring completion.

We evaluated the method using U.S. equity market stream data and a six-job ELT pipeline in an Apache Hudi lakehouse. The stream input was ingested for about three hours, totaling about 1.6×10^8 records (about 18 GB), and additional master data was preloaded.

The results highlight that subset selection is effective when dispersion is severe. In an observed execution of an upstream aggregation job, selecting 4 files out of 19 reduced scan size from 1497 MiB to 13 MiB, and increased marginal-benefit from 0.668 to 0.929. In contrast, in other jobs under our setting, subset selection did not appear: shrinking the prefix reduced execution time only slightly while staleness reduction decreased, so the best marginal-benefit was achieved by processing all incremental files.

At the pipeline level, we used the integrated staleness metric P and compared Partitioned-Max-Benefit with Random scheduling and Max-Benefit. The measured values were $P_{\mathrm{PMB}} = 73{,}875{,}680$, $P_{\mathrm{Max}} = 77{,}336{,}752$, and $P_{\mathrm{Random}} = 85{,}474{,}258$. Thus, Partitioned-Max-Benefit reduced P by 13.6% compared to Random and by 4.5% compared to Max-Benefit in this experiment.

[Conclusions] This thesis shows that file dispersion can limit the effectiveness of incremental queries: even "incremental" ELT runs may touch many files, making full-input processing inefficient for freshness optimization. Partitioned-Max-Benefit addresses this by selecting a cost-effective subset of incremental files per execution based on marginal-benefit, while keeping candidate enumeration tractable via a prefix restriction and maintaining consistent reflected-time semantics through t_j^{done}.

Our Apache Hudi implementation and market-data experiments demonstrate that subset-aware scheduling can reduce overall pipeline staleness under limited resources, especially in stages where incremental updates are dispersed across many files. Future work includes analyzing the conditions under which subset selection is most effective, scaling to larger clusters and higher concurrency, and extending the approach to other lakehouse systems.

[Reference]

[1] L. Golab, T. Johnson, and V. Shkapenyuk, "Scalable Scheduling of Updates in Streaming Data Warehouses," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(6), 1092–1105, 2012.

[2] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics," *CIDR*, 2021.

[3] Apache Hudi Documentation: Incremental Query and Timeline Metadata, accessed 2026.

[Keywords] Lakehouse, Apache Hudi, incremental query, ELT pipeline, scheduling, data freshness, staleness, marginal benefit