Performance Evaluation of Oracle APEX Reporting Regions: Analysis of Filtering and Sorting Performance

Ivan Pastierik, Michal Kvet
University of Žilina
Žilina, Slovakia
pastierik2@stud.uniza.sk

Abstract— **This paper presents a performance evaluation of filtering and sorting operations in Oracle APEX reporting regions, focusing on the Classic Report, Interactive Report, and Interactive Grid components. These regions differ in architectural design and in how processing is distributed between server-side SQL execution and client-side JavaScript models, which directly influences interaction responsiveness. The experiments were conducted using a normalized aviation dataset derived from the OpenFlights repository and deployed in an Oracle Autonomous AI Database environment. Performance measurements were collected through Oracle APEX Debug Mode to capture precise AJAX refresh timings and interaction response durations. Both standard SQL queries and intentionally delayed queries were evaluated to examine the impact of database latency on user interactions. In addition, alternative optimization strategies were analyzed, including client-side filtering and sorting of the currently displayed page and full client-side data loading via the Interactive Grid model. The results show that server-side operations scale predictably with data volume but are highly sensitive to query execution time, whereas client-side techniques significantly improve interaction speed for small and medium datasets at the cost of increased memory usage and longer initialization time for large datasets. The findings highlight the trade-offs between responsiveness, scalability, and resource consumption, and provide practical guidance for designing performance-efficient Oracle APEX applications under varying workload conditions.**

I. Introduction

Oracle Application Express (APEX) is a low-code development platform designed for building database-centric web applications directly on top of Oracle Database technology [1]. It provides a declarative development model in which user interfaces, data access logic, and application behavior are defined through metadata rather than extensive procedural programming [2]. Because Oracle APEX operates natively inside the database engine, it enables tight integration between SQL execution, security mechanisms, and application rendering, making it particularly suitable for enterprise dashboards, analytical tools, and transactional systems [3].

A fundamental architectural element in Oracle APEX is the region, a modular page component responsible for rendering content and managing user interaction [4]. Among the available region types, reporting components such as Classic Report, Interactive Report, and Interactive Grid are used for tabular data presentation [4]. These components differ in their internal implementation, especially in how they distribute processing between server-side SQL execution and client-side JavaScript logic [5].

Interactive capabilities such as filtering, sorting, aggregation, and inline editing enhance usability but introduce additional processing overhead. In many cases, filtering and sorting operations trigger asynchronous AJAX requests that result in re-execution or transformation of the underlying SQL query. Consequently, the overall responsiveness of an Oracle APEX application depends not only on database performance but also on the architectural characteristics of the selected reporting region [6].

From a database perspective, query performance is influenced by relational design, normalization level, indexing strategy, and join complexity [7]. However, in low-code environments such as Oracle APEX, performance also depends on framework mechanisms including region refresh cycles, client-side model updates, and data transfer between server and browser [5]. Despite the practical importance of these aspects, systematic empirical evaluation of filtering and sorting performance in Oracle APEX remains limited. While a few studies have investigated Oracle APEX performance under multi-tenant cloud workloads [8] and assessed broad optimization techniques for Oracle APEX applications [9], this study focuses specifically on systematic analysis of filtering and sorting latency across different reporting region architectures, thereby extending and complementing previous research.

This paper presents a structured performance analysis of filtering and sorting operations across major Oracle APEX reporting regions. The experiments are conducted using a normalized aviation dataset derived from the OpenFlights repository [10]. The application is deployed in an Oracle Autonomous AI Database environment to ensure stable and reproducible execution conditions [11]. Both standard and intentionally delayed SQL execution scenarios are examined to evaluate how Oracle APEX handles slower database responses and how region architecture influences interaction latency.

The results provide quantitative insight into how filtering and sorting scales with increasing data volume and how different implementation strategies affect usability. The findings contribute to practical recommendations for designing Oracle APEX applications that maintain responsiveness even when handling large datasets or operating under constrained query performance conditions.

II. Description of Oracle APEX Reporting Regions and the JavaScript Programming Interface

This section introduces the primary reporting region types: Classic Report, Interactive Report, and Interactive Grid, and outlines their structural and functional differences. It also provides an overview of the Oracle APEX JavaScript programming interface, which enables developers to extend the behavior of these regions, control data refresh cycles, and improve application responsiveness. Understanding the internal architecture of Oracle APEX reporting regions and their client-side APIs forms the foundation for subsequent performance measurements and optimization analysis.

A. Oracle APEX Reporting Regions

Oracle APEX structures application pages using modular building blocks known as regions, which encapsulate both presentation logic and data interaction behavior [1]. Regions can render static content, charts, forms, or tabular query results, and they define how information retrieved from the database is displayed and manipulated within the browser. In data-oriented applications, reporting regions are particularly significant because they mediate the interaction between SQL result sets and end users [4].

A reporting region in Oracle APEX is fundamentally driven by a SQL query whose result is transformed into an HTML-based representation during page generation [4]. Depending on the selected region type, the rendering process may rely entirely on server-side output generation or combine server-side data retrieval with client-side JavaScript processing. This architectural distinction directly influences performance characteristics, especially when handling filtering, sorting, and large datasets. Studies comparing server-side and client-side rendering in web frameworks illustrate how shifting work between server and client affects latency and user experience, highlighting the relevance of architectural differences in performance analysis [12].

The Classic Report represents the most lightweight tabular reporting mechanism. It renders query results directly into HTML using predefined templates during page generation. The component does not maintain a persistent client-side data model, and most processing occurs on the server. However, advanced features such as dynamic filtering or sorting require additional configuration or custom logic, often implemented through dynamic actions or JavaScript extensions [4].

The Interactive Report expands the capabilities of the Classic Report by incorporating built-in end-user customization features. It allows runtime modification of the dataset presentation, including filtering conditions, sorting criteria, grouping, aggregation, and saved report views. These operations are typically executed through asynchronous requests that cause the underlying SQL query to be reprocessed with modified predicates or ordering clauses [4]. While this approach enhances analytical flexibility, it introduces additional server-side processing cycles and AJAX communication overhead.

The Interactive Grid further extends reporting functionality by integrating data editing and rich client-side interaction. Unlike simpler report types, it maintains a structured JavaScript data model in the browser that represents the fetched result set [5]. This hybrid architecture combines SQL-based data retrieval with client-side model management, enabling inline editing, multi-row updates, validation, and advanced navigation features. Although this design improves responsiveness once data is loaded, it requires additional initialization time due to model construction, JSON processing, and widget binding [5].

From an architectural perspective, these three region types illustrate progressively increasing reliance on client-side execution. The Classic Report emphasizes server-side rendering, the Interactive Report balances server processing with dynamic refresh mechanisms, and the Interactive Grid introduces a full client-side data model layered on top of database-driven queries. These differences are central to understanding the filtering and sorting performance analyzed in this study.

B. JavaScript APIs in Oracle APEX

Oracle APEX provides a comprehensive JavaScript Application Programming Interface (API) that enables developers to extend the declarative capabilities of the platform with dynamic, client-side logic [5]. The JavaScript API acts as an integration layer between the browser environment and the Oracle APEX runtime engine, allowing programmatic interaction with regions, page items, buttons, and server processes. Through this interface, developers can manipulate data models, trigger asynchronous refresh operations, control rendering behavior, and implement advanced client-side functionality without requiring full page reloads. Such capabilities are particularly important in performance-sensitive applications, including dashboards, data grids, and analytical interfaces [1].

A central entry point into the client-side API is the *apex.region* namespace, which provides standardized access to regions defined on a page. Each region is uniquely identified by its Static ID, enabling developers to retrieve a JavaScript handle to the corresponding region object. Once obtained, the region object exposes methods such as *.refresh()*, *.call()*, and *.widget()*. The *.refresh()* method allows re-execution of the region's underlying SQL query and dynamic re-rendering of its contents without reloading the entire page, thereby improving efficiency in interactive scenarios. The *.call()* method invokes internal region-specific functionality, while *.widget()* exposes the underlying jQuery UI widget instance, granting deeper access to advanced components such as the Interactive Grid [5].

For complex tabular components like the Interactive Grid, Oracle APEX exposes a client-side data abstraction known as the Grid Model [5]. This model can be accessed through the region's widget interface and represents the dataset currently loaded in the browser. It maintains both row-level data and associated metadata required for rendering and interaction management. Two important internal attributes of the model are data and index. The *_data* array stores the JavaScript objects representing individual rows currently available in the grid, while index maintains a mapping between record identifiers and their positions within the data structure. These internal collections directly influence which records are displayed, how they are ordered, and how filtering or sorting operations modify the visible dataset. Inspecting these attributes provides insight into the client-side behavior of the grid and is particularly useful for performance experimentation.

By default, the Interactive Grid implements incremental data loading to reduce initial page rendering time [4]. Only the subset of rows required for the current page view is retrieved during

initialization. To override this behavior, the *fetchAll()* method can be used to load the complete dataset into the client-side model [5]. This method retrieves all available records from the server and populates the *_data* structure accordingly. While this approach enables full client-side filtering and sorting without additional server communication, it increases memory consumption and prolongs the initial loading phase, especially when handling large datasets. Therefore, its use requires careful consideration in performance-critical environments.

Programmatic refresh operations can target either the entire region or only its client-side data model. Invoking *apex.region("grid1").refresh()* re-executes the SQL query and synchronizes both server-side and client-side representations. In contrast, manipulating the internal model and calling grid-level refresh function *apex.region("grid1").widget().interactiveGrid('getViews').grid .refresh()* updates the rendered data without issuing a new database request if the data are already in client-side model [5].

Beyond region manipulation, the Oracle APEX JavaScript API includes additional namespaces that support dynamic application behavior. The *apex.item* namespace provides programmatic access to page item values and properties. The *apex.server* API enables asynchronous invocation of PL/SQL processes, facilitating background data exchange without full page reloads. The *apex.actions* namespace supports defining reusable actions that can be bound to interface elements such as buttons and menus, while *apex.message* and "apex.debug" provide mechanisms for user feedback and runtime diagnostics. Together, these APIs establish a rich client-side programming layer that enables performance optimization in interactive applications [5].

III. Dataset Description

The experimental performance evaluation uses a relational aviation dataset derived from the publicly available OpenFlights project [10]. The original data, distributed as CSV files containing airlines, airports, aircraft, and routes, was imported into Oracle Database staging tables and subsequently transformed into a normalized relational schema.

The schema was implemented in Oracle Autonomous AI Transaction Processing Database in Oracle Cloud, which hosts both the database and the Oracle APEX environment [11]. During transformation, data validation procedures ensured removal of inconsistent or incomplete records before inserting into the final structure. The resulting model shown in Fig. 1 follows third normal form (3NF) [7].

The *FL_ROUTE* entity links airlines with source and destination airports, while *FL_ROUTE_AIRPLANE* models the many-to-many relationship between routes and aircraft types. This relational design increases join complexity and reflects realistic transactional data structures commonly discussed in database systems literature [7].

To support scalability analysis, the dataset was extended through controlled synthetic generation of additional route records while preserving relational consistency. This enabled systematic testing of filtering and sorting performance across scalable datasets with sizes of 1 000, 10 000, 100 000, and 1 000 000 rows.

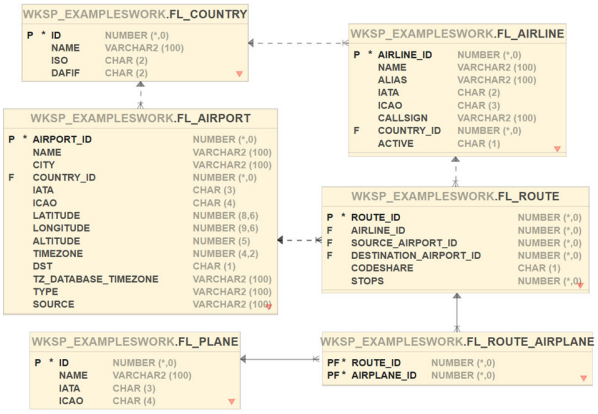

Fig. 1. Normalized OpenFlights data model used for performance analysis

IV. Methodology

The experimental design focuses on measuring the response time of filtering and sorting operations in Oracle APEX reporting regions. Because these interactions trigger asynchronous communication with the database, their performance depends on both region architecture and SQL execution behavior. To obtain reliable and comparable results, all tests were conducted in a controlled cloud environment using standardized region configurations and timing data captured through Oracle APEX Debug Mode.

A. Experimental Environment

The application and database were hosted on Oracle Cloud Infrastructure using an Autonomous Transaction Processing (ATP) database running Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 - Production Version 19.30.0.1.0 (Autonomous AI Database). The Oracle APEX runtime environment was version 24.2.13, deployed within the same cloud database instance.

This configuration ensures that both the database engine and the Oracle APEX execution layer operate within a managed and performance-isolated infrastructure. Because the Autonomous Database automatically handles indexing, resource scaling, and execution plan optimization, the environment provides a stable platform for comparative benchmarking.

All measurements were collected using Oracle APEX Debug Mode (Level 4). Debug Mode was selected because filtering and sorting operations in Oracle APEX are often executed asynchronously via AJAX requests. Standard browser timing tools do not capture the full server-side execution lifecycle of these operations. Debug Mode logs precise execution timestamps for page processing, region refresh, SQL execution, and AJAX callbacks, allowing accurate measurement of filtering and sorting latency.

Although Debug Mode introduces minor execution overhead due to logging, all tests were performed under identical conditions, ensuring that relative comparisons remain valid.

B. Data Retrieval Strategy

To evaluate how Oracle APEX handles filtering and sorting operations, two variants of the data source were tested:

1) Standard Query: The first query represents a typical relational workload joining multiple normalized entities:

```
SELECT r.route_id,
al.name AS airline,
sa.name AS source_airport, sc.name AS
source_country,
da.name AS dest_airport, dc.name AS
dest_country,
p.name AS airplane
FROM fl_route r
JOIN fl_airline al ON r.airline_id =
al.airline_id
JOIN fl_airport sa ON r.source_airport_id =
sa.airport_id
JOIN fl_country sc ON sa.country_id = sc.id
JOIN fl_airport da ON r.destination_airport_id
= da.airport_id
JOIN fl_country dc ON da.country_id = dc.id
JOIN fl_route_airplane ra ON r.route_id =
ra.route_id
JOIN fl_plane p ON ra.airplane_id = p.id
```

This query joins eight normalized tables and represents a moderately complex analytical workload. It was used as the baseline for measuring filtering and sorting responsiveness under realistic relational conditions.

2) Artificially Delayed Query: To simulate slow database response scenarios, a second variant of the same query was implemented inside a PL/SQL function that introduces a fixed delay before returning the SQL statement:

```
DBMS_LOCK.SLEEP(10);
return q'~
SELECT r.route_id,
al.name AS airline,
sa.name AS source_airport, sc.name AS
source_country,
da.name AS dest_airport, dc.name AS
dest_country,
p.name AS airplane
FROM fl_route r
JOIN fl_airline al ON r.airline_id =
al.airline_id
JOIN fl_airport sa ON r.source_airport_id =
sa.airport_id
JOIN fl_country sc ON sa.country_id = sc.id
JOIN fl_airport da ON r.destination_airport_id
= da.airport_id
JOIN fl_country dc ON da.country_id = dc.id
JOIN fl_route_airplane ra ON r.route_id =
ra.route_id
JOIN fl_plane p ON ra.airplane_id = p.id
~';
```

The inclusion of *DBMS_LOCK.SLEEP(10)* introduces a deterministic 10-second delay prior to query execution. This setup was designed to evaluate how Oracle APEX behaves when the underlying SQL source responds slowly.

The rationale for testing the delayed query is to demonstrate that filtering and sorting operations in Oracle APEX are executed by re-submitting the full source SQL statement (or function returning SQL) to the database. If filtering or sorting operations were performed purely on the client side, the artificial delay would not be observed during subsequent interactions. By measuring the latency during filter and sort actions, it becomes possible to confirm whether Oracle APEX delegates these operations to the database engine.

This approach also allows assessment of application usability under degraded database performance conditions, providing insight into end-user experience when query execution is slow.

C. Region Configuration

All tested regions were configured in their default state, with only minimal adjustments applied to avoid introducing confounding variables. Each page contained exactly one reporting region so that its behavior could be observed in isolation. Pagination was enabled for all regions, with 1 000 rows displayed per page to ensure consistent data volume during filtering and sorting operations. The Static ID property was uniformly set to “MAIN”, allowing precise identification of region refresh events during debugging and measurement.

No client-side caching mechanisms were enabled, and no custom JavaScript overrides or performance optimizations were introduced. Sorting behavior remained in its default configuration, and filtering relied exclusively on the native mechanisms provided by each region type. Apart from the explicitly tested filtering and sorting operations, no additional dynamic actions or enhancements were implemented. This controlled configuration ensured that the measured response times reflect the inherent implementation characteristics of each region type rather than the influence of external tuning or customization.

The experimental evaluation covered four reporting configurations across four dataset scales, resulting in sixteen distinct scenarios. For the Classic Report, two filtering approaches were tested: Faceted Search and Smart Filters. These were evaluated separately to compare the performance of declarative filtering mechanisms available for non-interactive report regions. In addition, the Interactive Report and Interactive Grid were tested using their built-in filtering and sorting capabilities.

Each of these four configurations was executed against datasets containing 1 000, 10 000, 100 000, and 1 000 000 rows. For every dataset size, filtering and sorting operations were triggered under identical structural and environmental conditions to ensure comparability across region types.

D. Measurement Procedure

Filtering and sorting performance was evaluated by initiating user interactions and measuring the elapsed time between the action trigger and the completion of the corresponding region refresh. All timing data were extracted from Oracle APEX Debug Mode logs, specifically from AJAX request entries associated with region refresh execution. Each experimental scenario was executed multiple times under identical conditions, and average response times were calculated to minimize the influence of transient effects such as caching behavior, network variability, or temporary database load fluctuations.

The experimental design considered several independent variables, including region type, dataset volume, operation type (filtering or sorting), and query execution mode (standard or delayed). The primary dependent variable was interaction response time, defined as the duration between user action and full completion of region refresh. By maintaining identical SQL structure, dataset composition, and execution environment

across all tests, the methodology isolates the impact of region architecture and database execution behavior on filtering and sorting performance.

V. Results – Standard Query Execution

Table I summarizes the measured filtering and sorting response times for all sixteen scenarios executed using the standard SQL query. The overall distribution of results across individual scenarios is visualized in Fig. 2, while aggregated comparisons by dataset size are presented in Fig. 3.

TABLE I. FILTERING AND SORTING TIMES FOR STANDARD QUERY

Index	Scenario	Dataset Size (Rows)	Filtering Time (s)	Sorting Time (s)
1	Classic Report Faceted Search	1 000	0,27	0,43
2	Classic Report Smart Filters	1 000	0,26	0,43
3	Interactive Report	1 000	0,24	0,31
4	Interactive Grid	1 000	0,28	0,64
5	Classic Report Faceted Search	10 000	0,31	0,47
6	Classic Report Smart Filters	10 000	0,3	0,47
7	Interactive Report	10 000	0,27	0,33
8	Interactive Grid	10 000	0,37	0,65
9	Classic Report Faceted Search	100 000	0,59	0,65
10	Classic Report Smart Filters	100 000	0,58	0,65
11	Interactive Report	100 000	0,44	0,47
12	Interactive Grid	100 000	0,76	0,8
13	Classic Report Faceted Search	1 000 000	1,8	2,22
14	Classic Report Smart Filters	1 000 000	1,75	2,22
15	Interactive Report	1 000 000	1,64	2,41
16	Interactive Grid	1 000 000	2,02	2,64

As shown in Table I and Fig. 2, interaction times increase consistently with growing dataset volume across all region types. For small datasets (1 000 rows), filtering operations range between 0.24 s (Interactive Report) and 0.28 s (Interactive Grid). Sorting operations vary from 0.31 s (Interactive Report) to 0.64 s (Interactive Grid). On this scale, all configurations provide quick response, and performance differences are negligible from an end-user perspective.

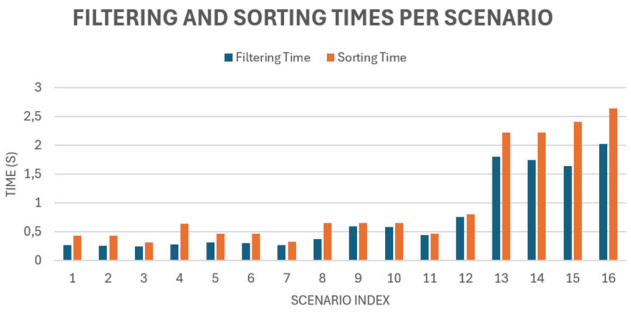

Fig. 2. Chart displaying filtering and sorting times per scenario for standard query

When dataset size increases to 10 000 rows, moderate growth is observed. Filtering times remain below 0.4 s for all regions, and sorting times stay below 0.7 s. The Interactive Report consistently demonstrates the lowest interaction times, as also visible in Fig. 3, where its bars remain below those of other region types across both filtering and sorting operations.

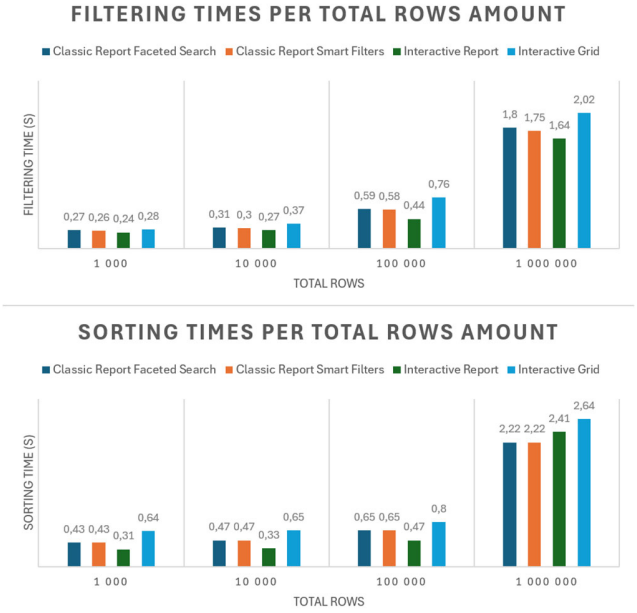

Fig. 3. Charts displaying filtering and sorting times per total rows amount for standard query

At 100 000 rows, performance divergence becomes more pronounced. Filtering times reach approximately 0.58–0.59 s for Classic Report configurations, 0.44 s for Interactive Report, and 0.76 s for Interactive Grid. Sorting times follow a similar pattern, with Interactive Grid reaching 0.80 s, while Interactive Report remains below 0.50 s. The separation between region types is clearly illustrated in Fig. 3, where the Interactive Grid shows steeper growth compared to other configurations.

The largest dataset (1 000 000 rows) highlights scalability differences most clearly. Filtering times increase to 1.75–1.80 s for Classic Report variants, 1.64 s for Interactive Report, and 2.02 s for Interactive Grid. Sorting operations exhibit even higher costs: 2.22 s for both Classic Report configurations, 2.41 s for Interactive Report, and 2.64 s for Interactive Grid. As visible in Fig. 3, sorting consistently requires more time than filtering at this scale, suggesting additional database reordering overhead during refresh operations.

Two important patterns can be observed. First, Classic Report with Faceted Search and Smart Filters demonstrates nearly identical performance across all dataset sizes, confirming that both filtering approaches rely on comparable database-level execution mechanisms. Second, the Interactive Report provides the most stable scalability profile, maintaining the lowest or near-lowest interaction times throughout all experiments. In contrast, the Interactive Grid exhibits the highest sensitivity to dataset growth, particularly for sorting operations, likely due to additional client-side model synchronization and rendering processes.

Overall, the combined evidence from Table I, Fig. 2, and Fig. 3 confirms that filtering and sorting operations scale predictably with dataset size and remain within acceptable usability thresholds even at one million rows. However, architectural differences between region types significantly influence responsiveness, with Interactive Report offering the most balanced trade-off between functionality and interaction latency under standard query conditions.

VI. Results – Delayed Query Execution

To verify whether filtering and sorting operations re-execute the underlying SQL source, a delayed query variant was introduced by inserting "DBMS_LOCK.SLEEP(10)" before returning the SQL statement. This artificial delay also allowed precise observation of how many times the query is executed during a single interaction.

The measured results are summarized in Table II, while the distribution per scenario is illustrated in Fig. 4 and the aggregated comparison by dataset size in Fig. 5.

TABLE II. FILTERING AND SORTING TIMES FOR DELAYED QUERY

Index	Scenario	Dataset Size (Rows)	Filtering Time (s)	Sorting Time (s)
1	Classic Report Faceted Search	1 000	20,25	20,43
2	Classic Report Smart Filters	1 000	20,31	20,43
3	Interactive Report	1 000	10,36	10,44
4	Interactive Grid	1 000	10,32	10,74
5	Classic Report Faceted Search	10 000	20,44	20,56
6	Classic Report Smart Filters	10 000	20,42	20,56
7	Interactive Report	10 000	10,41	10,46
8	Interactive Grid	10 000	10,57	10,84
9	Classic Report Faceted Search	100 000	20,69	20,74
10	Classic Report Smart Filters	100 000	20,71	20,74
11	Interactive Report	100 000	10,56	10,6
12	Interactive Grid	100 000	11	10,99
13	Classic Report Faceted Search	1 000 000	21,9	22,49
14	Classic Report Smart Filters	1 000 000	21,93	22,49
15	Interactive Report	1 000 000	11,78	12,57
16	Interactive Grid	1 000 000	12,31	12,73

A clear structural difference between region types immediately emerges. For both Classic Report configurations (Faceted Search and Smart Filters), filtering and sorting times are consistently around 20–22 seconds across all dataset sizes. In contrast, Interactive Report and Interactive Grid show response times around 10–12 seconds.

Because the artificial delay is exactly 10 seconds, these values directly reveal execution behavior. The approximately 20-second response time of Classic Reports indicates that the SQL returning function is executed twice during a single filtering or sorting action. Debug logs confirm this sequence: first, Oracle APEX invokes the function to describe the query structure and retrieve column metadata. Subsequently, it executes the generated component SQL to fetch the result set. Since the delay is embedded inside the function that returns the SQL text, both invocations incur the 10-second pause, resulting in a total delay of roughly 2 × 10 seconds.

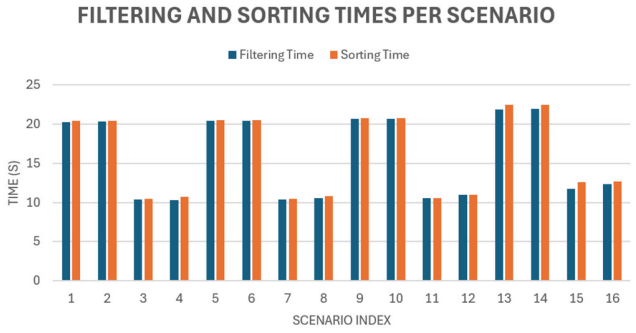

Fig. 4. Chart displaying filtering and sorting times per scenario for delayed query.

This behavior is clearly visible in the debug trace. The function containing *DBMS_LOCK.SLEEP(10)* is executed once during the metadata description phase and again during final query execution. Consequently, each interaction in Classic Report effectively doubles the imposed delay.

In contrast, Interactive Report and Interactive Grid show response times slightly above 10 seconds, indicating that the delayed SQL function is executed only once per interaction. The additional 0.3–2.5 seconds observed beyond the base 10-second delay correspond to normal region refresh overhead, and rendering initialization. The difference between Interactive Report and Interactive increases with dataset size, reflecting the higher client-side synchronization cost of the Interactive Grid.

Another important observation is that dataset size has only a marginal effect under delayed conditions. Whether 1 000 or 1 000 000 rows are processed, total interaction time is dominated by the artificial 10-second database delay. The incremental growth visible in Fig. 5, especially at 1 000 000 rows, represents genuine SQL processing and rendering overhead, but it remains secondary compared to the imposed wait time. This confirms that filtering and sorting operations in all tested region types are database-driven rather than purely client-side operations.

The comparison between standard query results and delayed query results therefore provides strong empirical evidence that Oracle APEX filtering and sorting mechanisms re-execute the full SQL source for each interaction. However, execution strategy differs between region types: Classic Report performs an additional metadata evaluation step that results in double invocation of the SQL returning function, whereas Interactive Report and Interactive Grid execute it only once per refresh cycle.

From a usability perspective, this distinction becomes critical in environments where queries are inherently slow. Under high-latency or computationally expensive conditions, Classic Reports may exhibit significantly longer interaction times due to repeated SQL evaluation, while Interactive Reports and Interactive Grids maintain comparatively lower response times.

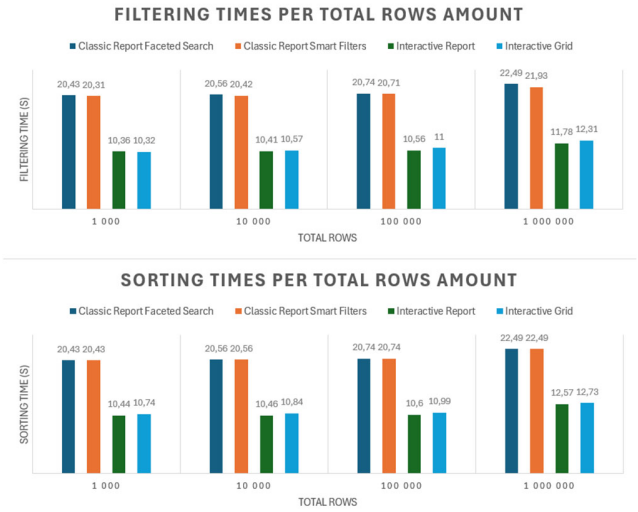

Fig. 5. Charts displaying filtering and sorting times per total rows amount for delayed query.

VII. Mitigation Strategies for Slow SQL Queries

When the underlying SQL query cannot be further optimized at the database level, alternative architectural strategies must be considered to preserve acceptable interaction responsiveness. Based on the experimental evaluation, three principal approaches were analyzed: materializing query results into a temporary table, client-side filtering and sorting of the currently displayed page, and full client-side filtering and sorting of all loaded data using the Interactive Grid data model.

A. Materializing Results in a Temporary Table

The first approach consists of executing the expensive query once and storing its result in a temporary or staging table. Subsequent filtering and sorting operations are then performed against this precomputed dataset rather than re-executing the full source query.

This strategy eliminates repeated execution overhead, which is particularly beneficial in cases where the query includes complex joins, analytical functions, or external service calls. In environments like the delayed query experiment, where a fixed execution latency dominates interaction time, materialization would effectively reduce filtering and sorting from approximately 10–20 seconds to sub-second database operations.

However, this solution introduces trade-offs. It increases storage requirements, requires session-level isolation or cleanup mechanisms, and may lead to stale data if the underlying tables change frequently. Therefore, it is most appropriate for analytical dashboards, reporting snapshots, or workloads where real-time consistency is not mandatory.

B. Client-Side Filtering and Sorting of the Current Page

The second approach shifts filtering and sorting logic entirely to the browser, but only for the rows currently displayed on the page. This was implemented using standard JavaScript DOM manipulation and array filtering and sorting operations applied to rendered table rows.

The results demonstrate extremely fast interaction times across all dataset scales. As shown in Fig. 6, filtering remained approximately 0.01 seconds and sorting approximately 0.03 seconds, even under delayed-query conditions. Importantly, these times are unaffected by the 10-second artificial database delay, because no additional SQL execution is triggered during interaction.

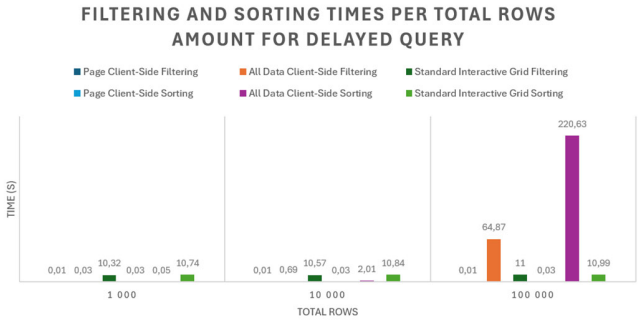

Fig. 6. Speed difference of filtering and sorting (delayed query, client-side variants)

This approach provides excellent responsiveness because operations are performed on a small in-memory subset (1 000 rows per page). Computationally, filtering and sorting scale linearly with the number of visible rows, resulting in negligible processing time at this scale.

The primary limitation is functional scope. Since only the currently displayed page is processed, filtering does not affect rows outside the active pagination window. Consequently, this method does not provide globally correct results across the full dataset. It is therefore suitable primarily for quick client-side refinements, exploratory analysis, or UI responsiveness enhancement rather than strict analytical correctness.

C. Full Client-Side Filtering and Sorting of All Data

The third strategy loads the entire dataset into the Interactive Grid model using „model.fetchAll()" and performs filtering and sorting directly on the client-side data structure. Unlike page-level filtering, this method preserves global correctness across all records.

For small datasets (1 000 rows), this approach performs very efficiently. Filtering requires only 0.03 seconds and sorting 0.05 seconds. Even with the delayed query, interaction times remain near-zero once data is fully loaded.

However, scalability limitations become evident as dataset size increases. At 10 000 rows, interaction times grow to 0.69 seconds for filtering and 2.01 seconds for sorting, with loading time exceeding 24 seconds. At 100 000 rows, performance degrades dramatically: filtering requires over 64 seconds and sorting exceeds 220 seconds, as shown in Fig. 6. Loading time also increases substantially and is over 75 seconds. For 1 000 000 rows, total loading time surpasses 457 seconds, making the approach impractical for large datasets.

These results reflect inherent computational complexity. Client-side filtering operates in O(n) time, while sorting approaches O(n log n). When n becomes large, browser memory constraints and JavaScript execution overhead dominate performance. Furthermore, transferring very large

datasets to the client introduces significant network and rendering overhead, as illustrated in Fig. 7.

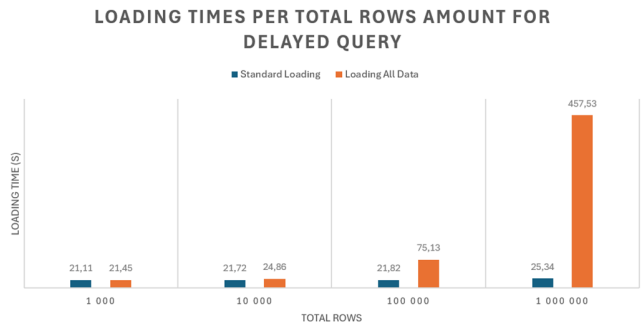

Fig. 7. Comparison of loading times (loading page vs. loading all data).

D. Comparative Evaluation

The comparative results highlight three distinct performance regimes:

- Database-driven filtering ensures correctness and scalability but becomes unusable when SQL execution is slow.
- Page-level client-side processing provides near-instant interaction but sacrifices global dataset consistency.
- Full client-side processing eliminates repeated database calls but becomes computationally infeasible beyond moderate dataset sizes.

Fig. 6 and Fig. 7 clearly demonstrate that client-side page filtering offers the best responsiveness under slow-query conditions, while full client-side loading is viable only for relatively small datasets. Materialization at the database level represents the most balanced and scalable mitigation strategy for large-scale data.

E. Practical Recommendations

When SQL optimization is not possible, the following hierarchy of solutions is recommended:

- Use temporary or materialized tables when correctness across the entire dataset is required and data volume is large.
- Use page-level client-side filtering for highly interactive dashboards where immediate responsiveness is more important than global filtering accuracy.
- Use full client-side data loading only for small datasets (≤10,000 rows), where memory and computational overhead remain manageable.

These findings demonstrate that performance bottlenecks in Oracle APEX applications can often be mitigated not only through database tuning but also through architectural adjustments that redistribute processing between server and client layers.

VIII. Conclusion

This paper presented a systematic evaluation of filtering and sorting performance in Oracle APEX reporting regions, focusing on the Classic Report, Interactive Report, and Interactive Grid components. Using a normalized aviation dataset derived from the OpenFlights project and deployed in an Oracle Autonomous AI Database environment, the experiments compared server-side and client-side interaction strategies under varying data volumes and query execution conditions.

The results demonstrate that filtering and sorting operations in Classic Reports and Interactive Reports are predominantly executed on the server side, as each interaction triggers re-execution or transformation of the underlying SQL query. When query execution is fast, response times remain within sub-second ranges even for moderately large datasets. However, when the SQL source is artificially delayed, interaction latency increases proportionally, clearly confirming the dependence of these region types on database execution time. Classic Report interactions may incur additional overhead due to repeated metadata processing before query execution.

The Interactive Grid exhibits more complex behavior due to its hybrid architecture combining server-side data retrieval with a client-side JavaScript model. In its default configuration, filtering and sorting also rely on server-side refresh operations. However, when client-side techniques are applied, interaction latency can be reduced to near-instantaneous levels for small and medium-sized datasets. This confirms that shifting workload to the browser can significantly improve perceived responsiveness when database execution time is the primary bottleneck.

Nevertheless, full client-side loading introduces substantial memory and initialization overhead as dataset size increases. While loading all data into the client model enables fully local filtering and sorting, it becomes impractical for very large datasets due to excessive loading time and browser resource consumption. The experiments therefore highlight a trade-off between initial loading cost and subsequent interaction speed.

Overall, the findings indicate that filtering and sorting performance in Oracle APEX is strongly influenced by region architecture and the distribution of computation between server and client. When SQL optimization is limited or query latency cannot be reduced, alternative strategies such as temporary materialized result sets or selective client-side processing may improve usability. However, these techniques must be applied with respect to dataset size and memory constraints.

Future work may extend this evaluation toward hybrid optimization strategies, including adaptive client-side caching, incremental loading mechanisms, and predictive prefetching techniques. These approaches have the potential to reduce sorting and filtering response times to below two seconds. Further investigation into indexing strategies, query plan stability, and Oracle APEX internal refresh cycles could also contribute to a deeper understanding of performance behavior in large-scale Oracle APEX applications.

Acknowledgment

This paper was supported by the **VEGA 1/0192/24** project - Developing and applying advanced techniques for efficient processing of large-scale data in the intelligent transport systems environment.

REFERENCES

[1] A. Png and H. Helskyaho, *Extending Oracle Application Express with Oracle Cloud Features*. New York: Apress, 2022.

[2] E. Sciore, *Understanding Oracle APEX 20 Application Development*, 3rd ed. New York: Apress, 2020.

[3] P. Cimolini, *Oracle Application Express by Design: Managing Cost, Schedule, and Quality*. Cham: Springer, 2017.

[4] Oracle Corporation, Oracle APEX documentation, Web: https://docs.oracle.com/en/database/oracle/application-express/.

[5] Oracle Corporation, Oracle APEX JavaScript API reference, Web: https://docs.oracle.com/en/database/oracle/application-express/24.1/aexjs/.

[6] S. R. Keshireddy, "Extending Oracle APEX for Large-Scale Multi-Form Workflows with Decoupled PL/SQL Logic and Asynchronous Processing Layers", *in Proc. 2025 International Conference on Next Generation Computing Systems (ICNGCS)*, Aug. 2025, pp. 1–8.

[7] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, 7th ed. Boston: Pearson, 2015.

[8] A. Syed, "Performance Analysis of Oracle APEX Applications in Multi-Tenant Cloud Environments", *International Scientific Journal of Engineering and Management*, vol. 4, Apr. 2025, pp. 1–9.

[9] A. Syed, "Optimizing Performance in Oracle APEX Applications: Techniques and Benchmarks", *Journal of Artificial Intelligence & Cloud Computing*, vol. 4, Jul. 2025, pp. 1–10.

[10] OpenFlights, Airport, airline and route data, Web: https://openflights.org/data.php.

[11] Oracle Corporation, Oracle Autonomous AI database, Web: https://www.oracle.com/autonomous-database/.

[12] V. Jain, "Server-Side Rendering vs. Client-Side Rendering: A Comprehensive Analysis", *International Journal of Innovative Research and Creative Technology*, vol. 7, Apr. 2021, pp. 1–5.