Transactional Panorama: A Conceptual Model for User Perception in Analytical Visual Interfaces

Dixin Tang, Indranil Gupta†, Aditya G. Parameswaran
UC Berkeley | † UIUC
{totemtang,adityagp}@berkeley.edu,indy@illinois.edu

ABSTRACT

Data analysis tools empower users to organize many analysis results in a single visual interface, like a dashboard. When the involved datasets are large, it is time-consuming to refresh all of the visualizations shown on one screen when the underlying data changes. We tackle the problem of presenting *fresh and coherent* results to a user while preserving *interactivity*. Coherence involves two properties across space and time: *consistency* (across multiple visualizations) and *monotonicity* (across multiple results on one visualization). Interactivity is typically achieved by ensuring *visibility*: a user can always read and interact with any analysis result. Existing tools make fixed choices on maintaining parts of these properties and sacrificing others that could have been desired by a user. We take a principled way to reason about the property space and propose *transactional panorama*, a conceptual model that adopts transactions to jointly model the behavior of the system refreshing the analysis results and the user interacting with the results. We formally define monotonicity, visibility, and consistency, along with new performance metrics cognizant of the user's semantics in visual interfaces. Then, we theoretically find all of the possible combinations of these properties and define the relative performance across them to reveal the fundamental trade-offs. Using transactional panorama as a lens, we discover three new property combinations that are not considered in existing tools. We build a system, *Fader*, which enables a user to choose appropriate properties and performance trade-offs for their specific use case. Our experiments on a Business Intelligence tool demonstrate significant performance difference across different property combinations, which verifies the values of the newly discovered property combinations and the model that enables these options.

1 INTRODUCTION

Many data-centric tools empower users to visually organize, present, and consume multiple data analysis results in a single interface, such as a dashboard. Each such analysis result is represented on this interface as a scalar value, table, or visualization, and is computed using the source data or other analysis results, in turn, as *views*. This pattern appears in a variety of contexts:

Visual analytics or BI tools like Tableau [12] or PowerBI [8] empower users to embed visualizations on a dashboard, each of which corresponds to a SQL query on an underlying database;

Spreadsheet tools such as Microsoft Excel and Google Sheets allow users to add derived computation in the form of spreadsheet formulae, visualizations, and pivot tables;

Data application builder tools such as Streamlit [10], Plotly [7], and Redash [9], enable users to efficiently develop interactive dashboards, employing computation done in Python UDFs and pandas dataframe functions, and SQL;

Lens name	Example Tools	Monotonicity	Visibility	Consistency
Globally-Consistent Fully-Blocking (GCFB)	MS Excel [6] Libre Calc [5] Tableau [12]	Yes	No	Yes
Globally-Consistent Partially-Blocking (GCPB)	Power BI [8] Superset [11] Dataspread [15]	Yes	No	Yes
Inconsistently Non-Blocking (ICNB)	Google Sheets [2]	Yes	Yes	No

Table 1: Properties maintained and sacrificed by existing tools

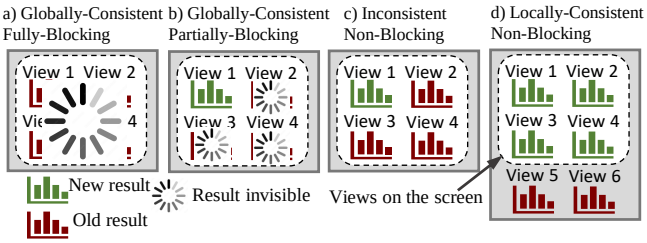


Figure 1: Visual examples of different lenses of refreshing views in a dashboard

Monitoring and observability tools such as Datadog [1], Kibana [4], and Grafana [3] empower users to make sense of their telemetry data and logs via a combination of automatically defined and customizable dashboard widgets.

In all of these contexts, there is *a network of views defined on underlying data, each of which is then visualized on an interface.* These views and the corresponding visualizations often need to be refreshed when the source data is modified. For example, a dashboard in a BI tool is refreshed with respect to regular changes to the underlying database tables (e.g., incorporating batches of new data). However, this refresh is rarely instantaneous, especially on large datasets. This represents a challenge, since the user is continuously exploring the visualizations as the refresh is performed: on the one hand, refreshing visualizations arbitrarily can be jarring to the user, since different visualizations on a given screen can be in different stages of being refreshed. On the other hand, not refreshing them in a timely manner can lead to stale results. So the key question we explore is the following: ***how do we allow users to continuously explore results in a visual interface, while ensuring that the results are not confusing or stale?***

Unfortunately, existing tools make fixed, and somewhat arbitrary decisions on how to address this question. For example, Excel [6], Calc [5], and Tableau [12] block the user from exploring the interface until all of the views are refreshed (Figure 1a). Other tools, like PowerBI [8], Superset [11], and Dataspread [15], improve on this approach by hiding (or greying) away any views that have not yet been refreshed, while still letting the user explore the other up-to-date views (Figure 1b). Yet other tools, like Google Sheets [2], opt for not hiding any views, and instead just progressively making

them available as they are refreshed—this approach has the downside of different results on the screen being in different stages of being refreshed, leading to incorrect insights (Figure 1c).

Transactional Panorama and Underlying Properties. In this paper, we instead introduce a formal framework, titled *transactional panorama*, to enable users and system designers to reason about the aforementioned question in a more principled manner. We adopt transactions to jointly model the system concurrently updating visualizations, with the user consuming these visualizations, over time and space (i.e., across screens). To the best of our knowledge, *transactional panorama is the first framework that leverages transactions to reason about correct user perception in visual interfaces.* In this setting, we define three key properties that we want to preserve: *consistency*, *monotonicity*, and *visibility* (the MVC properties). Consistency guarantees that the results displayed on the screen the user is viewing should be consistent with the same snapshot of source data [24, 41]—enabling correct derivation of relationships between results on the same screen. Monotonicity guarantees if a user reads the result for a view, any subsequent read will always return the same or more recent result (i.e., monotonic read [34])—ensuring that results never go "back in time". Visibility guarantees that the user can always explore the result of any view or visualization on the screen—instead of them being greyed out.

Concrete Property Combinations via Lenses. There are various mechanisms we can use to present results to the user in a visual interface, resulting in concrete selections for the aforementioned properties, that we call *lenses*[1]. Consider our examples from the previous paragraph; we list the corresponding three lenses in Table 1—GCFB, GCPB, and ICNB; while GCFB and GCPB opt for monotonicity and consistency, instead of visibility, ICNB opts for monotonicity and visibility, but not consistency. More broadly, in this work, we study the feasibility of different property combinations and lenses, and characterize their performance trade-offs. In particular, we throroughly explore the trade-off between *invisibility*, i.e., the times when the user is unable to interact with visualizations, and *staleness*, i.e., visualizations displayed to the user that haven't yet been refreshed, as displayed in Figure 2. For example, GCFB blocks the user from exploring the interface until all of the new results are computed, so it has high invisibility. But GCFB has zero staleness since it does not present stale results to the user. GCPB reduces GCFB's invisibility by presenting the newly computed results to the user whenever they are computed and also does not have stale results. ICNB, which sacrifices consistency, has higher staleness because the user can read stale results, but it does not include invisible views (i.e., zero invisibility).

Novel Property Combinations: Exploring the Trade-off. As we also show in Figure 2 (in red), we discover a number of novel lenses, resulting in new property combinations, and associated performance implications. Specifically, we introduce three new lenses: Globally-Consistent Non-Blocking (GCNB), Locally-Consistent Non-Blocking (LCNB), and Locally-Consistent Minimum-Blocking (LCMB), none of which are dominated by the lenses in existing tools. For example, LCNB always allows the user to inspect the results of any

[1]These are called lenses since they capture various instantiations of our transactional panorama framework.

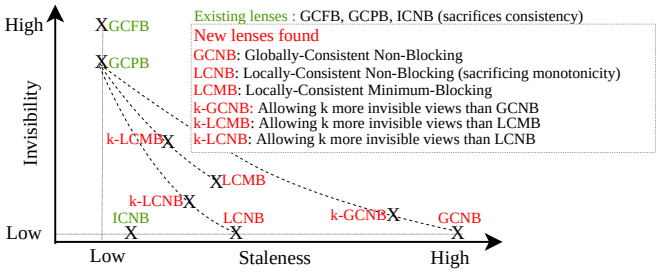


Figure 2: Trade-off between invisibility and staleness across different property combinations

visualizations (i.e., preserving visibility), and refreshes the visualizations on the screen when all of their new results are computed (i.e., preserving consistency). Figure 1 shows an example of LCNB, where the user can quickly read the new results on the screen (i.e., $View_{1-4}$) without waiting for computing the new results that are not on the current screen (i.e., $View_{5-6}$). LCNB can be used when a user wants to always see and interact with consistent results on the screen. However, as we prove later, LCNB needs to sacrifice monotonicity when the user explores different visualizations (e.g., by scrolling the mouse). In fact, as we demonstrate, one can achieve both consistency and visibility simultaneously only by either sacrificing monotonicity or suffering from high staleness. For the aforementioned new lenses, we further introduce k-relaxed variants (i.e., k-GCNB, k-LCNB, and k-LCMB), where k represents the number of additional invisible views allowed for the respective lenses. These lenses allow a user to gracefully explore the trade-off between staleness and invisibility.

Translating the Lenses to Practice. Simply identifying the aforementioned lenses and understanding how users may trade off staleness and invisibility is insufficient. Translate these lenses to practice in real data analysis and BI systems involve additional challenges:

(1) In a visual interface the user does not explicitly submit transactions as in traditional systems, but reads the views by looking at the screen. In addition, the user can read different subsets of views by scrolling to different screens. Therefore, it is not clear how we can adapt transactions to model user behavior, an aspect not considered in the transaction processing literature.

(2) Defining MVC properties is another challenge. A unique requirement in visual interfaces is that a user wants to quickly read the new results for some views before the system computes all of the new results. If we model an update along with refreshing the related views as a transaction (to preserve consistency), the user essentially wants to read the results of an *uncommitted transaction*. Defining monotonicity, visibility, and consistency for reading uncommitted results is not considered by traditional data management systems and needs to be addressed in our model.

(3) Finally, transactional panorama requires defining new performance metrics that consider the unique semantics of visual interfaces, including the time of a view being invisible or a visible view being stale when it is read by a user. With the new performance metrics defined, we also need to theoretically understand the relative performance across different property combinations and design new algorithms to optimize the performance.

We address these challenges in transactional panorama. We model reads and writes on the views as transactions. Specifically, we introduce a special read-only transaction to model a user reading the views on the screen. This transaction does not block waiting for the new result of a view if the new result is not computed, but immediately returns an invisible state, indicating the result is under computation. To model the fact that the user keeps "reading" the views, the read transaction is periodically issued to pull fresh results. On the other hand, a write on the source data along with refreshing the related views is modeled as a write transaction. Based on this abstraction, we define MVC properties for read transactions, and include a series of theorems, called *MVC Theorems*, to find the possible property combinations, some of which are newly discovered combinations. We further define two performance metrics, *invisibility and staleness*, and demonstrate the performance trade-offs across different property combinations. Invisibility represents the total time of the views read by users being invisible and staleness represents the total time of the views read by users being stale. We present the transactional panorama model in Section 2.

Based on transactional panorama, we build a system, Fader, intended for use with a BI tool. Fader allows a user to choose different lenses that maintain different properties with different performance. In Fader, we design efficient algorithms for processing read transactions and maintaining the properties selected by users, and propose optimizations to reduce invisibility and staleness for processing write transactions. The system design of Fader is presented in Section 3. We implement Fader in Superset, a popular open-source BI tool [11]. Our experimental results in Section 6 reveal the significant performance difference across different lenses, and demonstrate the values of the new lenses and the model that enables these options. In addition, the experiments show our optimizations can significantly reduce invisibility and staleness compared to the baseline (by up to 70% and 75%, respectively). Finally, we discuss the related work in Section 7 and conclude the paper in Section 8.

2 TRANSACTIONAL PANORAMA MODEL

We present our transactional panorama model in this section. We introduce necessary background in Section 2.1. Then, we discuss modeling reading/writing views using transactions in Section 2.2. Next, we formalize our model in Section 2.3, define MVC properties in Section 2.4, and evaluate the feasibility of various property combinations in Section 2.5. We introduce new property combinations and their k-relaxed variants in Section 2.6. Afterwards, we define performance metrics in Section 2.7, order the property combinations by the performance metrics in Section 2.8, and discuss selecting appropriate lenses for specific use cases in Section 2.9. Finally, we discuss extensions of the model in Section 2.10.

2.1 Background

View and View graph. In our context, we define a *view* to represent arbitrary computation, expressed in any manner, including relational operators, pandas dataframe expressions, spreadsheet formulae, or UDFs, taking other views and/or source data as input. A view graph is a directed acyclic graph (DAG) that captures the dependencies across views and source data, both represented as nodes in the DAG. Specifically, if a view n_i takes another view or source data n_j as input, we add a directed edge: $n_j \rightarrow n_i$. The *dependents* of n_j are defined as the views that are reachable from n_j in the view graph. For simplicity, we regard the source data as a special type of view that performs an identity function over the source data.

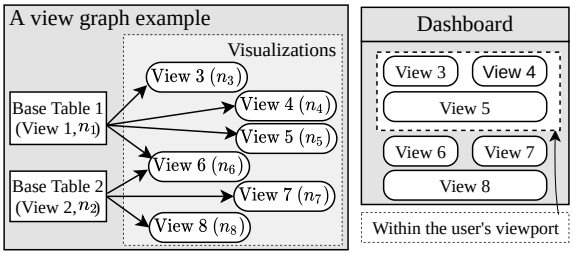


Figure 3: An example of a dashboard and its view graph

Figure 3 shows an example of a view graph for visualizations in a dashboard, where the source data are database tables and each view is defined by a SQL statement. There are two base tables: Base Table 1 and 2, also regarded as View 1 and 2, respectively. We use n_k to represent View k. View 3-6 (denoted n_{3-6}) and View 6-8 (denoted n_{6-8}) are the dependents of n_1 and n_2, respectively. They define the content for the visualizations in this dashboard.

Query result and Viewport. A *query result* represents the output of a view computed on a version of the source data. Each query result can then be rendered in an analytics tool as a value, a table, or a visualization, as part of a *dashboard*. We refer to all of these output modalities as *visualizations*. A dashboard may include many visualizations that cannot fit into a single screen. The rectangular area on the screen a user is currently looking at is the *viewport*. For example, in Figure 3 the viewport includes visualizations for views n_{3-5}. A user can change the viewport to explore different parts of a view graph.

Reads and writes on a view graph. Reading a view means reading the query result for the view. If the query result for a view is to be consumed by a user (e.g., on a dashboard), it needs to be (eventually) materialized. Future reads from the user can also use the materialized result. In Figure 3, we need to materialize results for n_{3-8} to support future reads by the user.

There are two types of writes in transactional panorama: *input writes* and *triggered writes*. An *input write* is from a user or an external system, and modifies the source data (e.g., a batch of new data inserted to a base table) or view graph definitions. In the following, we focus on input writes to the source data as it is the most common case of refreshing a dashboard. We discuss processing modifications to the view graph definitions in Section 2.10. The input write will trigger additional writes, called *triggered writes*, which compute new query results for the views that depend on the input write. For example, modifying the base table n_1 in Figure 3 triggers computing new query results for n_{3-6}.

2.2 Modeling the Interaction with a View Graph

We model a user's or an external system's interaction with a view graph as transactions, and logically associate each transaction with a unique timestamp that represents its submission time, with transactions being ordered by these timestamps. We focus on a single user setting as in most today's user-facing data analysis tools, such

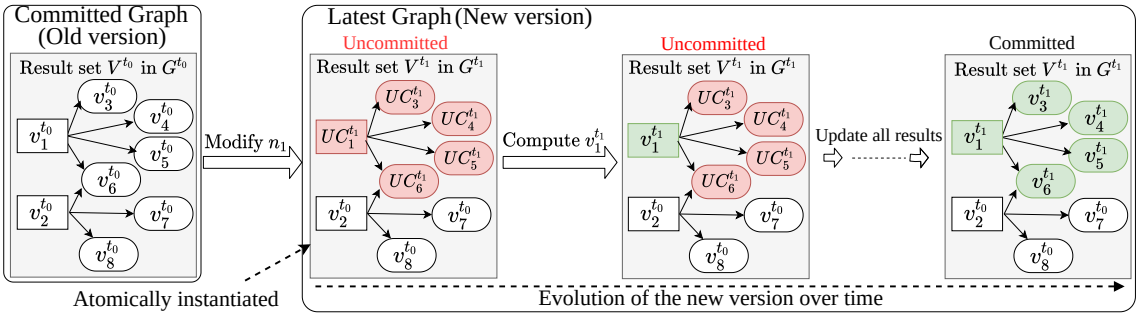


Figure 4: An example of creating a new version of view graph and computing the query result for each node

as Excel [6], Tableau [12], and PowerBI [8], and discuss the multiple user setting in Section 2.10. Our model has two types of transactions:

Write transaction. A write transaction is issued when a set of input writes on the view graph (e.g., modifications to a set of base tables) are submitted together to the system. A write transaction involves processing the input writes and recomputing the views that depend on the input writes. For example, in Figure 3 if n_1 is modified the write transaction will recompute n_{3-6}. The specific algorithm for maintaining the view graph and recomputing query results is orthogonal to our model, and, for example, can employ incremental view maintenance [13, 25, 36]. We focus on processing one write transaction at a time, which is typical in existing tools [6, 8, 11, 12], and discuss the case of multiple, simultaneous write transactions in Section 2.10.

Read transaction. transactional panorama models a user inspecting views in a single viewport as a read transaction. For example, in Figure 3, the read transaction involves reading views n_{3-5}. A unique property of this read transaction is that it does not block to wait for the requested view query results to be computed. A query result is said to be *under-computation* if it has not been computed yet. In this case, the read transaction returns an under-computation state such that the user cannot inspect and interact with the corresponding visualization in the visual interface. To simulate the effect of the user "looking at" the viewport, our model assumes the system periodically issues new read transactions to pull new results. The user may read different parts of the view graph by changing the viewport while the system continues to process writes. We note that the location of the user's viewport is known to the system throughout.

2.3 Formalization

We now introduce the aforementioned concepts in more detail and more formally. The view graph is logically multi-versioned, where a new version of the view graph $G^{t_i} = (E, N, V^{t_i})$ is instantiated by a write transaction with timestamp t_i (denoted as w^{t_i}). N represents the set of nodes (i.e., source data and views) in the graph and each edge $e = (n_{prec}, n_{dep}) \in E$ indicates that node n_{dep} has another node n_{prec} as input. V^{t_i} captures the results for the views in G^{t_i} and evolves as we process the write transaction w^{t_i}. So, at a given time, V^{t_i} may include three types of results: i) w^{t_i} has already finished computing the query result for a view $n_k \in N$ — this query result is represented as $v_k^{t_i}$; ii) w^{t_i} intends to compute the query result for a view n_k but has not done it yet — we represent this under-computation state as $\text{UC}_k^{t_i}$; and iii) a view n_k will not be updated by w^{t_i} and so the query result for n_k is from the last version of view graph. Figure 4 shows an example of computing a new version of view graph for a write transaction w^{t_1} that modifies n_1 in Figure 3. We see that creating a new version of view graph logically replicates the query results of the last version of the view graph, and marks all of the query results to be computed as UCs (in red). Each UC is replaced after the corresponding query result is computed (in green). We guarantee that a new version of view graph is atomically seen by read transactions via our concurrency control protocol, as will be discussed in Section 3. We call a version of view graph *committed* if its write transaction is committed; otherwise, this version is *uncommitted*. The initial version is G^{t_0}, which is modified by a sequence of write transactions $W = \{w^{t_1}, \cdots, w^{t_n}\}$, where w^{t_i} is submitted before w^{t_j} if $t_i < t_j$.

We also have a sequence of read transactions $R = \{r^{s_1}, \cdots, r^{s_m}\}$, where r^{s_i} is submitted before r^{s_j} if $s_i < s_j$. Recall that each read transaction corresponds to a single viewport and all of the views in it. We refer to the results returned by a read transaction r^{s_i} as H^{s_i}, which includes query results and/or UCs, the special indicator for under-computation states. H^{s_i} includes a result for each view in the user's viewport. If a read transaction returns a UC for a view, its corresponding visualization is marked as *invisible* in the dashboard. On the front end, this can be displayed in various ways: grayed out, a progress bar, a loading sign, etc. We use UC for $\text{UC}_k^{t_i}$ when k and t_i are clear from the context.

2.4 MVC Properties

We now formally define the so-called MVC properties for read transactions, motivated by user needs in analytical visual interfaces. First, the user consumes the results of read transactions in order: i.e., they consume the results of one read transaction before the next. Therefore, they expect to see *monotonically* newer results for each view, avoiding the confusion that the results seen "travel back in time". Second, in a user-facing dashboard, the notion of *visibility* helps ensure interactivity, which means the user can continuously explore the query results of different visualizations without interruption, while the system processes write transactions. Finally, *consistency* helps ensure that insights derived from multiple visualizations on a given viewport are computed from the same snapshot of source data [24, 38, 41]. We now describe each property in detail.

Monotonicity. Monotonicity means if a user reads a given version of a query result or UC for a view, any successive reads on the same view will return the same or more recent version of the query result or UC. Formally, monotonicity is defined as:

Definition 1 (Monotonicity). *A sequence of read transactions $R = \{r^{s_1}, \cdots, r^{s_m}\}$ maintains monotonicity if the following holds: for any view n_k that is read by any two transactions r^{s_i} and r^{s_j}, the timestamps of the returned results are t_p and t_q, respectively; we have $t_p \leq t_q$ if $s_i < s_j$.*

To maintain monotonicity, the system needs to track the timestamps of the results for each view returned by the recent read operations, as will be discussed in Section 3.

Visibility. This property says that for any view that is read by any read transaction, the system should not return an under-computation state, UC. Formally, visibility is defined as:

Definition 2 (Visibility). *A sequence of read transactions $R = \{r^{s_1}, \cdots, r^{s_m}\}$ maintains visibility if for the results H^{s_i} that are returned by any transaction r^{s_i}, we have UC $\notin H^{s_i}$.*

If the preserving visibility requirement is too stringent, the user may opt for partial visibility, where they accept a controlled number of UCs as a trade-off for reading fresher results, which we will discuss next.

Consistency. In our setting, consistency means that the results read by each read transaction belongs to a single version of the view graph. Consistency is formally defined as:

Definition 3 (Consistency). *Let H^{s_i} be the results read by r^{s_i}. We say r^{s_i} maintains consistency if there exists a version of view graph $G^{t_j} = (E, N, V^{t_j})$ such that $t_j \leq s_i$ and $H^{s_i} \subseteq V^{t_j}$.*

Intuitively, $t_j \leq s_i$ requires that a user read a version created by the write transactions that happen before r^{s_i}. The condition $H^{s_i} \subseteq V^{t_j}$ guarantees that the results returned belong to a single version of view graph. Consider a read transaction r^{s_1} that reads n_{3-5}. Say for G^{t_1} in Figure 3, we have computed $v_3^{t_1}$ but not $v_{4-5}^{t_1}$. If the returned results for r^{s_1} is $H^{s_1} = \{v_3^{t_1}, \text{UC}_{4-5}^{t_1}\}$, then r^{s_1} maintains consistency because H^{s_1} belongs to V^{t_1}.

Note that consistency in transactional panorama is different from the traditional Consistency (C) property in ACID for database transactions. C in ACID refers to the property that each transaction correctly brings the database from one valid state to another. In our context, consistency is more closely related to Isolation (I) in ACID, which defines when results of one transaction can be read by others. So our notion of consistency allows a read transaction to read uncommitted results from a concurrently running write transaction (e.g., reading the uncommitted G^{t_1}), but additionally maintains the semantics that the returned results correspond to a single version.

A follow-up question about preserving consistency is which version of view graph a read transaction should read. Specifically, since we process one write transaction at a time, a read transaction can choose between reading the last committed version of the view graph, which we call the *committed graph*, and the version that the latest write transaction is computing, which we call the *latest graph*. Depending on the version that is read, we define three types of consistency, which opt for different trade-offs between invisibility and staleness (recall that invisibility refers to the time when the views in the viewport are invisible, while staleness refers to the time when the returned query results are not consistent with the latest graph; both will be defined in Section 2.7). The first type of consistency is **Consistency-fresh** or C_f, which always reads the latest graph. C_f returns fresh results but suffers high invisibility. For the example in Figure 4, with C_f, read transactions always read G^{t_1} while we are processing the write transaction.

Another type of consistency, called **Consistency-committed** or C_c, always reads the most recently committed graph. C_c does not have invisible views, but the staleness of the returned query results could be high. In Figure 4, if C_c is used, read transactions cannot read G^{t_1} until we have computed all of the query results for G^{t_1} (i.e., $v_1^{t_1}$ and $v_{3-6}^{t_1}$).

We additionally introduce a type of consistency that lies between C_f and C_c. This type of consistency requires that a read transaction read the most recent version of the view graph that returns the minimum number of UCs for this transaction, which we call **Consistency-minimal** or C_m for short. With C_m, we would typically read the committed graph to avoid returning UCs when the new query results in the viewport are not yet computed. Once they are computed, we can read the latest graph to return fresh query results. Consider reading n_{3-5} in Figure 4. Initially, the read transactions will read G^{t_0} because G^{t_0} does not include UCs. After the new query results for n_{3-5} are computed, we will read G^{t_1} because reading G^{t_1} for n_{3-5} does not return UCs, and G^{t_1} is more recent than G^{t_0}. Note that when the user changes the viewport, the minimum number of UCs returned by a read transaction may not always be zero for C_m. As we prove next, adopting C_m may sacrifice visibility if we need to additionally maintain monotonicity.

2.5 Feasibility of Property Combinations

We now develop a series of theorems, that we call the MVC Theorems, to characterize the complete subsets of MVC properties that can be maintained together. Specifically, we define the possible property combinations involving each type of consistency (i.e., C_c, C_f, and C_m). The first two straightforward theorems establish the fact that always reading the committed graph provides monotonicity and visibility for free, while always reading the latest graph maintains monotonicity, but sacrifices visibility.

Theorem 1 (C_c). *Maintaining consistency-committed will also maintain monotonicity and visibility for read transactions.*

Proof. (Sketch) If consistency-committed is adopted, read transactions read a committed version of view graph, which does not include UCs. So visibility is preserved. Since consistency-committed requires reading the most recently committed version, whose timestamp advances monotonically, monotonicity is preserved. □

Theorem 2 (C_f). *Maintaining consistency-fresh will also maintain monotonicity for read transactions, but consistency-fresh and visibility cannot always be maintained.*

Proof. Consistency-fresh (or C_f) requires always reading the latest graph, whose timestamp monotonically advances. So both

C_f and monotonicity are maintained. In addition, always reading the latest graph can return UCs, violating visibility. □

Unfortunately, with C_m, we cannot have both monotonicity and visibility. The intuition is that if two consecutive read transactions involve overlapping views, the latter transaction needs to read the same or more recent version of view graph compared to the former one to maintain monotonicity. Therefore, the latter transaction may read the latest graph, which may include UCs and thereby violate visibility.

Theorem 3 (C_m-impossibility). *We cannot always simultaneously maintain monotonicity, visibility, and consistency-minimal for read transactions.*

Proof. (Sketch) We construct a counterexample where the three properties cannot be met together. We assume the initial graph is G^{t_0} and a user modifies the base table n_1 in Figure 4. This modification creates a write transaction w^{t_1} that updates n_1 and n_{3-6}, and generates a new version G^{t_1}. We further assume we have computed the new results $v_1^{t_1}$ and $v_{3-5}^{t_1}$, but not $v_6^{t_1}$.

Based on this setup, consider two consecutive read transactions r^1 (reading n_{3-5}) and r^2 (reading n_{5-7}), which correspond to the case that the user moves the viewport. To maintain consistency-minimal, r^1 will read G^{t_1} and return $v_{3-5}^{t_1}$ since G^{t_1} does not include UCs for r^1. Now we show the subsequent transaction r^2 cannot maintain the three aforementioned properties simultaneously. To maintain monotonicity and consistency-minimal, r^2 has to read G^{t_1}. This is because both r^2 and r^1 need to read n_5, and r^1 has already read $v_5^{t_1}$ in G^{t_1}. However, reading G^{t_1} violates visibility because r^2 needs to read n_6 but $v_6^{t_1}$ has not yet been computed for G^{t_1}. This example proves that monotonicity, visibility, and consistency-minimal cannot always be met together. □

Interestingly, if we sacrifice one property among the three properties, we can always maintain the other two.

Theorem 4 (C_m-possibility). *Transactional panorama can always maintain any two properties out of monotonicity, visibility, and consistency-minimal for read transactions.*

Proof. (Sketch) First, we can maintain monotonicity and visibility together for any read transaction because for each view read by this transaction, we can always return the most recent query result, without considering consistency. Second, we can also maintain visibility and consistency-minimal (or C_m). To achieve this, each read transaction reads the most recent version of view graph that has zero UCs for this transaction. Finally, we can maintain monotonicity and C_m together because for a read transaction we could always find a version of view graph that meets monotonicity (e.g., the latest one). Among all of the versions that maintain monotonicity, we choose the one that minimizes the number of UCs for this transaction and thus achieves C_m. □

2.6 Discovering New Property Combinations

Given the feasible property combinations, we now describe the different ways of presenting the results to the user while preserving a property combination, which we call *lenses*[2]. We first introduce the lenses from existing tools, and discuss the newly discovered lenses and their variants. For simplicity, we use M for Monotonicity and V for Visibility.

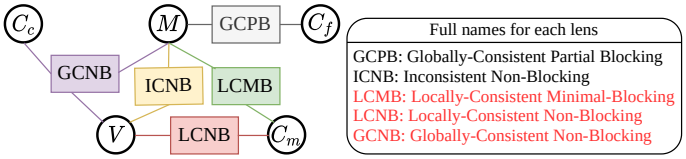


Figure 5: The possible property combinations and the corresponding lens covered in transactional panorama

Existing lenses:. The lenses that are adopted by existing tools include:

- GCPB: Globally-Consistent Partially-Blocking
- GCFB: Globally-Consistent Fully-Blocking
- ICNB: Inconsistent Non-Blocking

GCPB always presents any new query results that are consistent with the newly modified source data, or marks a view invisible if its query result has not been computed yet. This lens is adopted by Power BI [8], Superset [11], and Dataspread [15]. Its semantics are therefore equivalent to always reading the latest graph. So GCPB preserves M-C_f based on Theorem 2. GCFB marks all of the views invisible until the system has computed all of the new query results. This lens is adopted by Excel [6], Libre Calc [5], Tableau [12]. We don't consider it henceforth since it is dominated by GCPB, which means that GCPB maintains the same properties, and has lower invisibility and equal staleness compared to GCFB. ICNB allows a user to always inspect query results of any views and refreshes each view independently, which sacrifices consistency but preserves M-V based on Theorem 4. This lens is adopted by Google Sheets [2].

New lenses:. In addition to the existing lenses, transactional panorama includes three new lenses that correspond to the new property combinations not covered in existing tools:

- GCNB: Globally-Consistent Non-Blocking
- LCNB: Locally-Consistent Non-Blocking
- LCMB: Locally-Consistent Minimum-Blocking

GCNB adopts M-V-C_c from Theorem 1, and always reads and presents the query results in the recently committed graph to the user. LCNB adopts V-C_m from Theorem 3. It reads the recent version of view graph that does not have UCs for each read transaction, ensuring each viewport does not have invisible views and that the query results are consistent within the viewport. LCMB adopts M-C_m from Theorem 4. It reads the recent version of view graph that returns the minimum number of UCs for each read transaction and preserves monotonicity. Specifically, if reading either the committed or latest graph preserves monotonicity, LCMB chooses the version that has the minimum number of UCs for the read transaction. Otherwise, it reads the latest graph to preserve monotonicity, which may include UCs. We call these lenses *base lenses*, and their names and corresponding properties are summarized in Figure 5.

***k*-relaxed variants:.** As we will show in Figure 6, the three new lenses above are optimized to achieve low invisibility, but have high staleness. Therefore, we introduce their k-relaxed variants to allow for more invisible views to reduce staleness such that a user can gracefully explore the performance trade-off between

[2]We call these lenses since they help us capture various instantiations of our transactional panorama.

invisibility and staleness, as visualized in Figure 2. Specifically, k-GCNB, the variant of GCNB, will read the latest graph if this graph involves k or fewer UCs, while GCNB reads the latest graph only when it is committed. Similarly, k-LCNB, corresponding to LCNB, reads the most recent version of view graph that has k or fewer UCs for the read transaction, ensuring the viewport has k or fewer invisible views. k-LCMB, the variant of LCMB, needs to maintain monotonicity and consistency, and works as follows. If reading either the committed or latest graph preserves monotonicity, k-LCMB reads the recent version that has k or fewer UCs for the read transaction, similar to k-LCNB. Otherwise, k-LCMB reads the latest graph to preserve monotonicity.

2.7 Performance Metrics

With the different lenses defined, we now formally define the performance metrics: invisibility and staleness, for these lenses, and study their relative performance in Section 2.8. Invisibility represents the total time when the views read by a user are invisible. We adapt the metric from previous work [15] to our scenario of modeling reading the view graph as read transactions. We define I, the invisibility for a set of read transactions $R = \{r^{s_1}, \cdots, r^{s_m}\}$, as:

$$I(R) = \sum_{i=1}^{m-1} |H_{\text{UC}}^{s_i}| \times (Time(r^{s_{i+1}}) - Time(r^{s_i}))$$

$Time(r^{s_i})$ is the time when r^{s_i} returns results, where $H_{\text{UC}}^{s_i}$ represents the set of UCs in the returned results. So $|H_{\text{UC}}^{s_i}| \times (Time(r^{s_{i+1}}) - Time(r^{s_i}))$ represents the time when the views read by r^{s_i} stay invisible between two consecutive read transactions.

Staleness represents the total time when read transactions' returned query results are not consistent with the latest version of view graph. We use S to denote staleness for a set of read transactions $R = \{r^{s_1}, \cdots, r^{s_m}\}$. Say G^{t_i} is the latest version of view graph before the read transaction r^{s_i} starts, and say the returned query result by r^{s_i} for view n_k is $v_k^{t_j}$. S is defined as:

$$S(R) = \sum_{i=1}^{m-1} \sum_{v_k^{t_j} \in H_{qr}^{s_i}} \mathbb{I}[v_k^{t_j} \notin V^{t_i}] \times (Time(r^{s_{i+1}}) - Time(r^{s_i}))$$

Here, $H_{qr}^{s_i}$ represents the query results that are returned by r^{s_i}. $\mathbb{I}[v_k^{t_j} \notin V^{t_i}]$ is 1 if the result is stale (i.e., $v_k^{t_j}$ does not belong to the result set V^{t_i} of the latest version of view graph); otherwise, it is 0. So the inner summation represents the total time when the query results returned by r^{s_i} stay stale between two consecutive read transactions.

2.8 Performance Metrics: Guarantees

To understand the fundamental trade-offs between invisibility and staleness for different lenses, we now order them by the two performance metrics, as is summarized in Figure 6. The following analysis assumes the same write transaction, the same order of computing the new query results for the views involved in the write transaction, and the same sequence of read transactions for all lenses. Recall that we already assume the system processes one write transaction at a time, so the read transactions either read the committed or latest graph. We use the acronyms for the respective

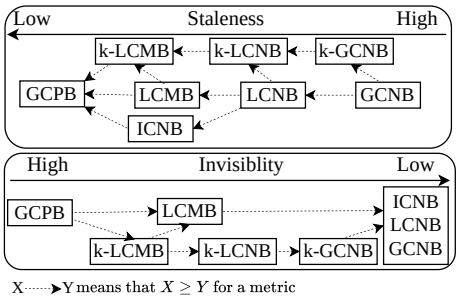


Figure 6: Summary of the orders across different lenses based on staleness and invisibility

lenses; the full names can be found in Figure 5. For simplicity, we use $S(A)$ and $I(A)$ to represent the staleness and invisibility for lens A, respectively.

We start with comparing GCPB and GCNB with other lenses. These two represent extreme points across the trade-off between staleness and invisibility since GCPB always reads the latest graph, while GCNB always reads the committed graph, and therefore sandwich other lenses.

Theorem 5. *$S(GCPB) \leq S(\text{all other lenses}) \leq S(GCNB)$, and $I(GCPB) \geq I(\text{all other lenses}) \geq I(GCNB)$.*

Proof. (Sketch) GCPB always reads the latest graph, so its staleness is zero and no larger than the other lenses, and its invisibility is no smaller than the other lenses (except for GCFB, which is excluded). Symmetrically, GCNB reads the latest graph only after all of the new query results are computed, so its staleness is no smaller than the other lenses, and its invisibility is no larger than the other lenses. □

Next, we use two theorems to order LCMB, LCNB, and ICNB by staleness and invisibility, respectively. Intuitively, LCMB always reads the same or more recent version than LCNB because if LCMB needs to read the latest graph to maintain monotonicity, LCNB may still read the committed graph; otherwise, both lenses read the same version of the view graph. So, LCMB has higher invisibility but lower staleness than LCNB. In terms of LCNB and ICNB, ICNB reads the new query result for a view whenever it is computed, but LCNB needs to wait for all the new query results in the viewport to be computed before reading them. So, LCNB has higher staleness than LCNB and they have the same zero invisibility. Finally, we find the order between LCMB and ICNB for invisibility depends on the workload, but LCMB has higher invisibility than ICNB since LCMB may return UCs while ICNB will not.

Theorem 6. *$S(LCMB) \leq S(LCNB)$, $S(ICNB) \leq S(LCNB)$, and the order between S(LCMB) and S(ICNB) depends on the workload.*

Proof. (Sketch) We prove $S(LCMB) \leq S(LCNB)$ by showing that LCMB always reads the same or more recent version than LCNB. Specifically, we show the two cases are true: 1) if LCMB reads the committed graph, then LCNB also reads the committed graph, and 2) if LCMB reads the latest graph, LCNB may not read the latest graph. The first case is true because if LCMB reads the committed graph, it means that the latest graph includes UCs for the read transaction and LCNB also reads the committed graph. The second case is also true because when LCMB reads the latest graph,

LCNB may still read the committed graph, sacrificing monotonicity in the process.

$S(ICNB) \leq S(LCNB)$ because ICNB reads the new query result for a view whenever it is computed, but LCNB needs to wait for all the new query results in the viewport to be computed before reading them.

The order between $S(ICNB)$ and $S(LCMB)$ depends on the configuration of the view graph, viewport, and read/write transactions. Consider when a viewport involves multiple views to update and all of the read transactions read this viewport. In this case, LCMB has higher staleness than ICNB because ICNB can read the new result for a view whenever it is computed but LCMB needs to wait for all of the new query results in the viewport to be computed. LCMB can also have lower staleness than ICNB. Say a user first waits for the views in the viewport to be up-to-date and then proceeds to examine other views. If any two consecutive read transactions have overlapping views, then after LCMB reads the latest graph for the first viewport, it will always read the latest graph for the rest of the viewports to preserve monotonicity. In this case LCMB has high invisibility, but low staleness. ICNB, on the other hand, does not always read the query results in the latest graph after the initial viewport and can have higher staleness than LCMB. □

Theorem 7. *$I(LCMB) \geq I(LCNB) = I(ICNB)$ for invisibility.*

Proof. (Sketch) LCNB and ICNB have the same zero invisibility since they are non-blocking. LCMB has larger or equal invisibility compared to LCNB and ICNB because LCMB may return UCs and does not always have zero invisibility. □

Our final two theorems will order each k-relaxed variant and their corresponding base lens, and the three k-relaxed variants relative to each other. Intuitively, a k-relaxed variant always reads the same or more recent version than the corresponding base lens due to the k additional UCs allowed, so it has no larger staleness and no smaller invisibility than the corresponding base lens. In addition, the orders across k-relaxed variants are the same as their corresponding base lenses since they allow the same number of UCs.

Theorem 8. *A k-relaxed variant has no larger staleness and no smaller invisibility than the corresponding base lens*

Proof. (Sketch) The k-relaxed variant always reads the same or more recent version than the corresponding base lens due to the k additional UCs allowed, and thus has no larger staleness and no smaller invisibility than the corresponding base lens. □

Theorem 9. *$S(k\text{–}LCMB) \leq S(k\text{–}LCNB) \leq S(k\text{–}GCNB)$ and $I(k\text{–}LCMB) \geq I(k\text{–}LCNB) \geq I(k\text{–}GCNB)$.*

Proof. (Sketch) To compare k-LCMB and k-LCNB, we show that k-LCMB will consistently read the same or more recent version of the view graph than k-LCNB. Specifically, if k-LCMB reads the committed graph, it means the latest graph includes more than k UCs for the viewport, so k-LCNB also reads the committed graph by definition. In addition, if k-LCMB reads the latest graph, k-LCNB may read the committed graph because k-LCNB does not need to preserve monotonicity. So we have $S(\text{k}\text{–}LCMB) \leq S(\text{k}\text{–}LCNB)$ and $I(\text{k}\text{–}LCMB) \geq I(\text{k}\text{–}LCNB)$. In terms of k-LCNB and k-GCNB, k-LCNB reads the latest graph when this graph has k or fewer UCs for the read transaction, but k-GCNB can read the latest graph only when the latest graph has k or fewer UCs overall. So we have $S(\text{k}\text{–}LCNB) \leq S(\text{k}\text{–}GCNB)$ and $I(\text{k}\text{–}LCNB) \geq I(\text{k}\text{–}GCNB)$. □

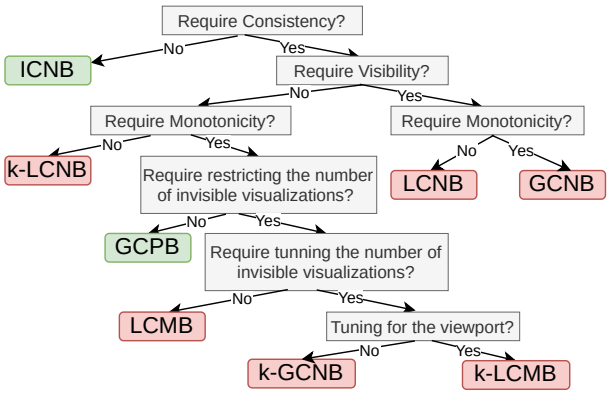


Figure 7: Summary of the proper selection of lenses

2.9 Selecting Lenses for Different Use Cases

The right lens depends on the user needs and specific use cases. We now provide a guidance for selecting the appropriate lens, summarized as a decision tree in Figure 7, where the existing lenses are marked as green boxes and the new lenses are marked as red. If the user does not require consistency across multiple visualizations, ICNB is the best choice since it maintains both monotonicity and visibility, and has low staleness. Otherwise, we check whether visibility is required. If yes, we choose between LCNB and GCNB, depending on the whether the user wants to additionally maintain monotonicity. If visibility is not required, we additionally check whether monotonicity is required. If not, we use k-LCNB to reduce staleness and invisibility. Note that k-LCNB subsumes LCNB. If the user does not want to tune the k value, they can adopt LCNB. On the other hand, if monotonicity is demanded, we check whether the user needs to restrict the number of invisible visualizations in the dashboard. If not, which means the user wants to see the new query results without worrying about invisibility, then GCPB is adopted. Otherwise, we further check whether the user wants to tune the k value. If not, LCMB is adopted. If yes, k-GCNB or k-LCMB is adopted depending on whether the user wants to tune the k value with respect to the viewport or the dashboard overall.

2.10 Extensions and Discussion

We now discuss extending our transactional panorama model to support modifying view graph definition, concurrent write transactions, and multiple users, and the UI designs.

Supporting modifying view graph definition. So far, we did not consider modifying view graph definitions. The most common modification is to apply a filter on the source data to explore the different results on different subsets of source data. For example, cross-filtering [26, 30] allows a user to apply a filter from one visualization to others and is commonly used in visual dashboards. transactional panorama can be extended to support this type of

modification by modeling applying a filter on a base table as modifying the content of the base table which is supported in our current model. For other types of modifications to the view graph definitions (e.g., deleting or inserting a visualization), which are expected to be rare, we support processing them using GCNB. That is, the user can read the new results after we have committed the write transaction.

Supporting concurrent write transactions. We also support concurrent write transactions in a single user setting, which means a user or an external system can submit a write transaction while the previous one is not finished. Here, each new write transaction creates a new version of view graph as discussed in Section 2.3 and write transactions are committed with respect to the order they are submitted. The definitions of MVC properties for read transactions do not change in this scenario. For example, for consistency-minimal, a read transaction returns the results of the version that has the minimal UCs for this transaction.

Supporting multiple users. transactional panorama can be easily extended to support multiple users reading the same dashboard. Here, each user can choose different MVC properties and we process each user's read transactions based on the selected properties separately. We leave the case of supporting concurrent writes from multiple users to future work.

UI designs. Instantiating transactional panorama requires new UI designs in the dashboard, such as helping the user identify the version of view graph they are looking at and effectively demonstrate the multiple choices of lenses to the user in the dashboard. For example, to differentiate two versions (for the case of processing one write transaction at a time), we can annotate the visualization that belongs to the committed graph and requires to be updated with a special indicator, such as a progress bar on the side. Prior work also studies the UI designs for differentiating multiple versions [38], which can be leveraged in transactional panorama if multiple concurrent write transactions exist. However, the UI designs are not the focus of this paper and left for future work.

3 MAINTAINING MVC PROPERTIES

We now discuss how to maintain different property combinations for different lenses defined in transactional panorama. Specifically, we discuss the design of the view graph and auxiliary data structures (Section 3.1), and the algorithms for maintaining MVC properties separately (Section 3.2-3.3) and maintaining property combinations for each lens (Section 3.4). We assume a *ReadTxn manager* responsible for processing read transactions, and another *WriteTxn manager* responsible for processing write transactions. The ReadTxn and WriteTxn managers are assumed to run on separate threads to enable concurrent execution of the two types of transactions.

3.1 View Graph and Auxiliary Data Structures

We maintain a multi-versioned view graph. Each node stores a list of results, called *result list*, where a result could be a query result or UC, and created by a new write transaction. Recall that a UC is a place-holder for the corresponding query result. Each node in the view graph is associated with a latch to synchronize concurrent reads/writes to its result list.

We additionally maintain an auxiliary table, MetaInfo, to store the timestamps of the last committed and latest view graphs (denoted as t_s and t_c, respectively), and the number of UCs for the latest graph (denoted as c_{uc}). The quantities t_s, t_c, and c_{uc} are maintained by the WriteTxn manager and will be used by the ReadTxn manager to preserve the properties specified by the user. We also include a latch to synchronize concurrent accesses to the MetaInfo table, which means any access to MetaInfo needs to acquire this latch.

3.2 Maintaining Consistency

We now discuss preserving three types of consistency: C_c, C_f, and C_m, and will discuss preserving monotonicity and visibility separately in the next subsection. Intuitively, maintaining consistency for a read transaction means this transaction can read the recently committed and latest view graphs. However, traditional concurrency control protocols, such as 2PL or OCC [16], do not apply here because they do not support the type of consistency that reads an uncommitted version of view graph (i.e., C_f and C_m). To maintain consistency, we process a read transaction into two steps:

1) Atomically find the timestamps for the last committed and latest view graphs (i.e., t_c and t_s in MetaInfo)
2) Read the versions of view graph for t_s and t_c.

Step 1) is done correctly via the latch on the MetaInfo table. Step 2) requires that the view graphs for t_s and t_c exist, which is done by the WriteTxn manager. Step 2) additionally requires an algorithm for reading a version of view graph for given a timestamp, which is done in the ReadTxn manager. We now discuss the designs of ReadTxn and WriteTxn managers for maintaining consistency.

WriteTxn manager. The WriteTxn manager processes a write transaction w^{t_i} in three steps:

1) Create a new version of view graph for w^{t_i}
2) Compute the query results for the views involved in w^{t_i} and update the view graph with the new results
3) Commit w^{t_i}

Step 1) guarantees that the timestamp of the latest view graph, t_s, exists. Specifically, the WriteTxn manager creates a new version of view graph by appending UCs to the result lists of the nodes that w^{t_i} needs to update[3]. Then it atomically updates t_s with the timestamp of the running write transaction w^{t_i}, and c_{uc}, the number of UCs for the latest graph, in MetaInfo. Step 2) computes the query result and replaces the corresponding UC for each view, and updates c_{uc}. It leverages a scheduler to decide the order of computing the query results to reduce invisibility and/or staleness, which we discuss in Section 4. Step 3) commits w^{t_i} by updating the timestamp of the last committed version (i.e., t_c) with t_i, which guarantees that the version of view graph for t_c exists.

ReadTxn manager. The ReadTxn manager uses timestamps t_s and t_c to read the last committed and latest view graphs. Depending on the properties that need to be maintained, the ReadTxn manager decides which version to read, which is discussed in the next subsection. Here, we present the algorithm for reading a version of view graph. Assuming a transaction r^{s_j} needs to read G^{t_i}, the intuition is that for each view read by r^{s_j}, we read the recent query result/UC

[3]For GCNB and ICNB, which do not need to read the uncommitted version, we can skip generating UCs as an optimization

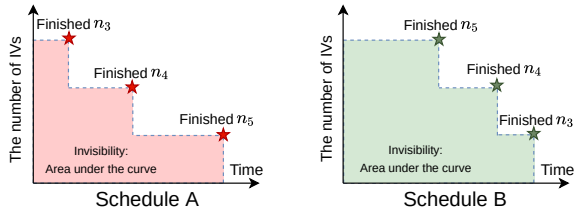


Figure 8: The impact of different schedules on invisibility

whose timestamp is no larger than t_i. The reason is that we have two possible cases if a query result/UC for a node n_k belongs to G^{t_i}: 1) n_k is or will be modified by w^{t_i}, in which case we have a query result/UC whose timestamp is t_i; 2) n_k is not modified by w^{t_i}, in which case the most recent query result whose timestamp is smaller than t_i belongs to G^{t_i}.

3.3 Maintaining Monotonicity and Visibility

To guarantee monotonicity, in the ReadTxn manager we maintain a table (denoted as *LastRead*) that stores the timestamps of the query results or UCs that are last read. Monotonicity requires that a read transaction reads the query results or UCs whose timestamps are no smaller than the corresponding timestamps in the LastRead table. To maintain visibility, we guarantee that the results returned by a read transaction do not include UCs.

3.4 Maintaining Property Combinations

For the lenses that need to maintain consistency, we read the MetaInfo and LastRead table to decide which versions of view graph to read. Specifically, to maintain M-C_f for GCPB we use t_s in MetaInfo to read the latest graph. Similarly, to preserve M-V-C_c for GCNB, we use t_c to read the last committed graph. For k-GCNB, we need to check whether the number of UCs for the latest graph (i.e., c_{uc} in MetaInfo) is smaller than k. If so, we read the version for t_s, otherwise, t_c is used.

Maintaining V-C_m for LCNB will read both the last committed and latest view graphs. Among the two sets of returned results, we choose to return the set of results that do not have UCs and correspond to the more recent version. Maintaining M-C_m for LCMB requires preserving monotonicity. This is done by checking the LastRead table to see whether reading the committed version violates monotonicity. If not, LCMB follows the same procedure of LCNB. Otherwise, LCMB will read the latest graph. k-LCNB and k-LCMB are processed similarly with k UCs relaxed.

For the property combination M-V (adopted by ICNB), which sacrifices consistency, we do not need to read MetaInfo. Instead, we directly read the view graph and return the most recent query result for each node involved in the read transaction.

4 WRITE TRANSACTION SCHEDULER

As mentioned in Section 3, the scheduler in the WriteTxn manager, which decides the order of computing the new query results for the views involved in a write transaction, significantly impacts the invisibility and staleness. We analyze two factors that impact the performance of the scheduler and design a scheduling algorithm that considers these factors to reduce invisibility and staleness.

Algorithm 1: Scheduling algorithm in the WriteTxn manager

```
1 N_w^{t_i} ← Topologically sort N_w^{t_i}
2 O ← Break N_w^{t_i} into topologically independent groups
3 for O^l ∈ O do
4     while O^l is not empty do
5         n_max ← arg max_{n_k ∈ O_l} P_k^{t_i} = D_k^{t_i} / Q_k^{t_i}
6         Update the view n_max
7         O^l ← O^l \ {n_max}
8     end
9 end
```

Factors that impact invisibility and staleness. Say a write transaction w^{t_i} updates a set of nodes $N_w^{t_i}$. Since a user can spend different amounts of time reading different views, the scheduler should prioritize updating the views that the user is more likely to read. In addition, different amounts of time are needed to compute query results for different views. Prioritizing updating the view that has the lowest execution time can reduce invisibility and staleness because it can quickly make a new query result visible, which is also observed in prior work [15]. We use an example to illustrate this. Here, say the write transaction w^{t_i} updates the base table n_1 and n_{3-6} in Figure 4, the views n_{3-5} are in the current viewport, and the viewport does not change. We additionally assume we need to maintain properties M-C_f (i.e., always reading the last created version) and want to minimize invisibility. Figure 8 shows two different schedules for n_{3-5}, where the x-axis plots the execution time while the y-axis plots the number of UCs. The invisibility is essentially the area under the curve: *the sum of the times of each view being invisible*. We see that while both schedules have the same execution time, Schedule A has lower invisibility than Schedule B because it always prioritizes computing the query result that has the least execution time. This heuristic can similarly reduce staleness for other property combinations (e.g., M-C_m).

Scheduling algorithm. We design a metric that considers the aforementioned factors and decides the priority of updating a view. First, our metric considers the likelihood that a user will read a view n_k during the execution of w^{t_i}. The likelihood is estimated using the total time when n_k is in the viewport since w^{t_i} has started (denoted as $D_k^{t_i}$). This statistics is collected by the ReadTxn manager. Second, our metric considers the execution time of computing the new query result for a view n_k. We use the cost estimated by the database as a proxy (denoted as $Q_k^{t_i}$). The metric for the priority of a view n_k is defined as: $P_k^{t_i} = \frac{D_k^{t_i}}{Q_k^{t_i}}$, where a larger value of $P_k^{t_i}$ means a higher priority.

Algorithm 1 shows the scheduling algorithm. We first sort $N_w^{t_i}$ topologically, break them into topologically independent groups, and compute each group with respect to the topological order. That is, we should only compute query results for views whose precedents are updated. To schedule a view to be updated within a group, we compute $P_k^{t_i}$ for each yet computed view n_k in this group and choose the view with the highest $P_k^{t_i}$.

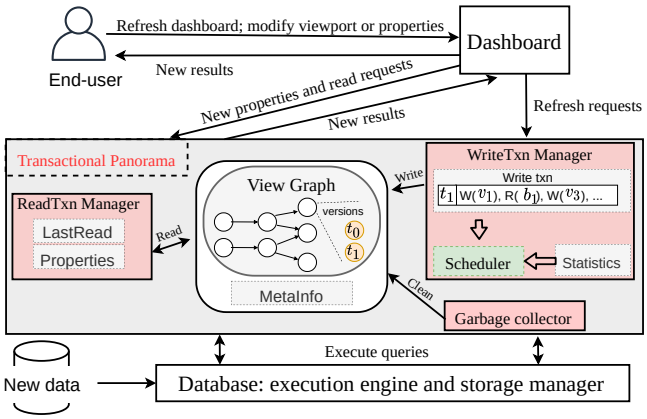


Figure 9: Overview of the System Implemented in Superset

5 PROTOTYPE IMPLEMENTATION

We now discuss implementing the transactional panorama model along with the algorithms for processing read and write transactions. We implement transactional panorama in Superset [11], a widely used open-source BI tool that has 47.3k stars on Github and similar functionality to Tableau [12] and PowerBI [8]. Superset provides a web-based client interface, where a user can define visualizations and organize them as part of a dashboard. Superset adopts a web server to process front-end requests and employ a database to store the base tables and compute the new query results for visualizations.

Figure 9 shows an overview of the system that implements transactional panorama in Superset. The user interacts with a dashboard by changing the viewport to explore different subsets of visualizations, selecting desired properties, and refreshing the dashboard with respect to changes to the underlying data. (The dashboard can also be configured to periodically refresh.) When the dashboard needs to be refreshed, it sends a refresh request to the WriteTxn manager, where each refresh request is interpreted as a write transaction. After a refresh request is sent, the dashboard will regularly send read requests to the ReadTxn manager to pull refreshed visualizations. Each read request includes the viewport information, which is leveraged by the ReadTxn manager to construct read transactions. When the system has finished the write transaction, the ReadTxn manager returns the new query results that have not yet read by the user to the dashboard to refresh all of its visualizations, after which, the dashboard stops sending new read requests. When the system is not processing a write transaction, the user can change the desired properties. Changing the properties on the fly is left for future work.

We store the base tables in the database and maintain the results of the derived visualizations along with the view graph in memory in the web server. We also include a garbage collector to clean up query results in the view graph that are guaranteed to not be read in the future. To safely clean up the useless query results, we maintain a timestamp, t_r, representing the oldest version of view graph that a read transaction is reading or will read in the future. With t_r, the garbage collector can safely remove the query results that belong to the older versions of view graph than t_r. We maintain t_r by updating it with t_c, the timestamp for most recently committed version of view graph, before each read transaction starts, because the oldest possible version a read transaction will read is the committed graph.

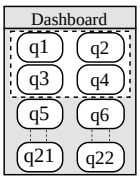
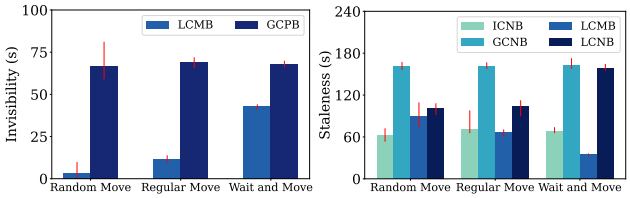


Figure 10: TPC-H dashboard

Configurations	Options	Default Value
Read behavior	{Regular Move, Wait and Move, Random Move }	Regular Move
Explore range	{22, 16, 10, 4}	22
Viewport size	{4, 10, 16, 22}	4

Table 2: Experiment configurations



(a) Invisibility **(b) Staleness**

Figure 11: Evaluations of different read behaviors

6 EXPERIMENTS

The high-level goal of our experiments is to characterize the relative benefits of different lenses given various workloads in order to help the user select the right lens for their needs and make appropriate performance trade-offs. Our experiments also demonstrate the value of the new lenses, which provide new trade-off points for the user to select, and evaluate the performance benefit and overhead of the optimizations for the write transaction scheduler. Our experiments address the following research questions:

1) [Workload impact on performance] How do different user behaviors and dashboard configurations impact the invisibility and staleness for the base lenses in transactional panorama? (Section 6.2)
2) [Performance benefit of new lenses] how much do the new lenses reduce invisibility while maintaining consistency compared to existing lenses? (Section 6.2)
3) [Trade-offs provided by k-relaxed variants] How do different k values impact the invisibility and staleness for the k-relaxed variants? (Section 6.3)
4) [Impact of optimized scheduling] How much does our write transaction scheduler decrease or increase invisibility and staleness? (Section 6.4)

6.1 Benchmark and Configuration

Benchmark. We build a dashboard based on the TPC-H benchmark. This dashboard includes 22 visualizations for all of the 22 TPC-H queries and runs on 1 GB of data, which is stored in PostgreSQL. This dashboard places two visualizations in a row, as shown in Figure 10. We test one refresh of the dashboard with respect to modifications to the base tables in the database since the relative performance across different lenses is independent of the number of refreshes. Specifically, we insert 0.1% new data to the tables Lineitem, Orders, and PartSupp, which triggers refreshing all of the

22 visualizations. One test ends when we have computed the new query results for all visualizations.

We build a test client to simulate different user behaviors and dashboard configurations. Similar to the web client of Superset, this client sends web requests to the web server to trigger a refresh (i.e., start a write transaction), configure the lens used for processing a refresh, and regularly pull refreshed results of visualizations in the viewport (i.e., start read transactions). We simulate three types of user behaviors in moving the viewport to read different visualizations, which we call the *read behavior*: 1) *Regular Move*: regularly moving the viewport downward or upward and reversing the direction if we reach the boundary of the dashboard; 2) *Wait and Move*: similar to the first one with the difference that it only moves the viewport after all of visualizations in the viewport are refreshed; and 3) *Random Move*: randomly chooses a viewport. For the three behaviors, the viewport is placed at the top of the dashboard at the beginning of each test and moved every 1 second. For the first two behaviors, each move changes the viewport by a row of visualizations. The test client can additionally vary the number of visualizations the user will inspect in the dashboard (denoted as *explore range*) during a test. We assume these visualizations are at the top of the dashboard. For example, explore range 4 means that the user will explore the visualizations for q_{1-4} in Figure 10 during a refresh. Our experiments also vary the number of visualizations in the viewport (denoted as *viewport size*) to evaluate how the relative sizes of the viewport and the dashboard impact invisibility and staleness. The experiment configurations are summarized in Table 2 and we use default configurations unless otherwise specified.

Configurations. The experiments are run on a t3.2xlarge instance of AWS EC2, which has 16 GB of memory and 8 vCPUs, and uses Ubuntu 20.04 as the OS. Our experiments use PostgreSQL 10.5 with default configurations. The time interval between two consecutive read transactions is set to be 100 ms to avoid overwhelming the web server, that is, the test client sends requests to pull refresh results every 100 ms. We run each test three times and report the mean number except for the tests that involve Random Move. For those tests, we run each 10 times and report the min, max, and mean.

6.2 Performance of Base Lenses

Takeaways: 1) The new lenses significantly reduce invisibility while preserving consistency compared to the existing lens GCPB; 2) A larger explore range increases the invisibility and staleness for all lenses except GCNB; 3) The staleness of all lenses except GCPB increases with a larger viewport size and converges to GCNB when the viewport size equals the dashboard size.

We evaluate the configurations in Table 2 for the base lenses.

Read behavior. Figure 11 reports the invisibility and staleness for the base lenses under different read behaviors. Each test reports the mean number and includes the min/max number as the error bar (i.e., the red line). We note that if the invisibility or staleness for a lens is zero, then that lens is not shown in the corresponding figure. To better see the trade-off between invisibility and staleness, we also plot the two metrics together in Figure 12 for Regular Move. We observe significant differences in invisibility and staleness for different lenses in Figure 11 and the new lenses (i.e., LCMB, LCNB, and GCNB) can significantly reduce invisibility while maintaining consistency compared to the existing lens GCPB. Specifically, GCPB has the highest invisibility because it always reads the latest graph, and LCMB, the lens that first reads the committed graph and switches to read the latest graph, can reduce invisibility by up to 95.2% compared to GCPB. LCMB reduces less invisibility in the case of Wait and Move because it spends a longer time on reading the latest graph. That is, after the first viewport, LCMB always reads the latest graph for the rest of the viewports due to the overlapping views across consecutive viewports. LCNB and GCNB have zero invisibility compared to GCPB. Recall that LCNB always reads the version of view graph that has zero UCs for the viewport to maintain consistency and visibility, but sacrifices monotonicity, and GCNB reads the last committed graph until all of the new query results are computed for the latest graph. ICNB also has zero invisibility, but sacrifices consistency.

On the other hand, GCNB has the highest staleness, but LCMB, LCNB, and ICNB can significantly reduce staleness. For example, LCMB reduces staleness by up to 78.9% compared to GCNB (i.e., the Wait and Move behavior). We note that LCMB reduces the staleness by sacrificing visibility while ICNB and LCNB need to sacrifices consistency and monotonicity, respectively. Overall, these results show that it is valuable to enable a user to have these options to make appropriate trade-offs. In addition, Figure 11b verifies the result that the order between ICNB and LCMB for staleness is undecided in Section 2.8 because we observe that the staleness of ICNB can be either higher or lower than LCMB in Figure 11b.

To better understand the behaviors of different lenses, we further report the number of invisible and stale views in the returned results of read transactions for Regular Move while we are processing the write transaction. For better exposition, we aggregate the read transactions that finish for every 2s and report the mean numbers in Figure 13. We note that the areas under the curve represent the invisibility/staleness in the respective figures. In Figure 13a, we see that GCPB initially returns many UCs as it reads the new version, and the number of UCs decreases as we compute more query results. Specifically, for the first 10s, a user sees more than 3 UCs on average, out of the 4 visualizations in the viewport. Therefore, this user almost cannot interact with the dashboard for the first 10s, significantly diminishing interactivity. LCMB, on the other hand, initially reads the committed graph to avoid invisible views. Then, it reads the latest graph (i.e., after 15s) to present fresh results to the user as GCPB does. Figure 13b shows the number of stale views over time. GCNB reads the same number of stale views as the viewport size (i.e., 4 in our test) until the last transaction. LCMB, LCNB, and ICNB can read the new version during the refresh, which reduces the staleness.

Explore range. This experiment evaluates the impact of varied explore ranges on different lenses. The results in Figure 14 show that we have smaller invisibility/staleness for GCPB, LCMB, LCNB, and ICNB when the user explores smaller number of visualizations because these lenses are more likely to read the new results for smaller explore ranges. The staleness for GCNB is independent of the value of explore range since it only refreshes the visualizations after all of the query results for the new version are computed.

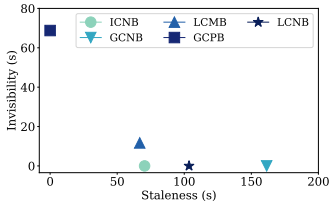
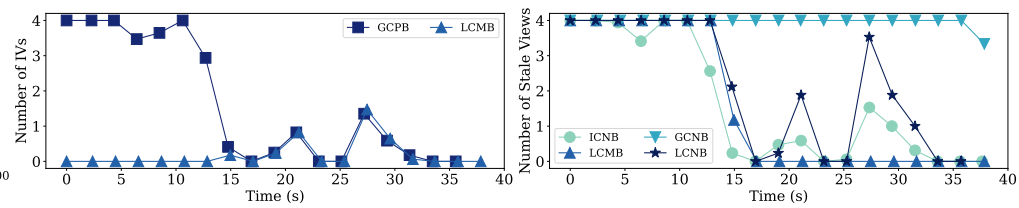
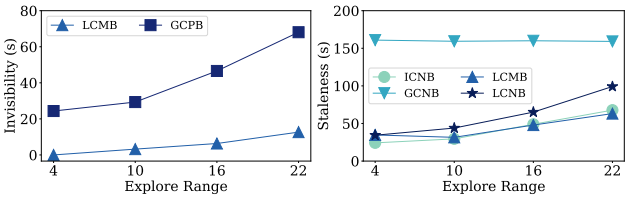


Figure 12: Performance trade-off for Regular Move



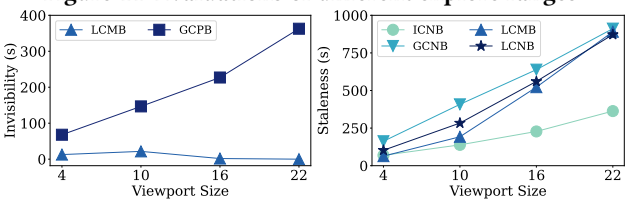
(a) Invisibility (b) Staleness

Figure 13: The number of invisible and stale views in the viewport during the refresh



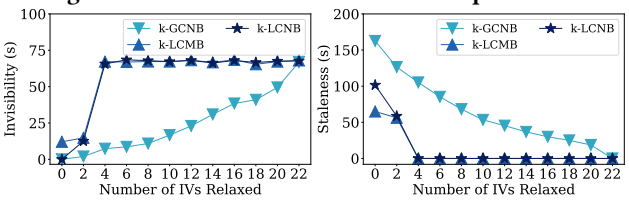
(a) Invisibility (b) Staleness

Figure 14: Evaluations of different explore ranges



(a) Invisibility (b) Staleness

Figure 15: Evaluations of different viewport sizes



(a) Invisibility (b) Staleness

Figure 16: Evaluations of varied k values

Viewport size. Figure 15 reports the results of varying the viewport sizes. We see that the invisibility for GCPB increases as the viewport size becomes larger because the user will read more invisible views for each read transaction. However, the invisibility for LCMB slightly increases and then decreases to zero. The reason for decreasing invisibility is that a larger viewport size pushes LCMB to wait longer to read the latest graph, which decreases invisibility. In an extreme case, when the viewport size covers the whole dashboard, LCMB's performance converges to GCNB, which has zero invisibility but the highest staleness, as shown in Figure 15b. We also see that the staleness for GCNB, LCMB, LCNB, and ICNB increases as they will read more stale views in one read transaction.

6.3 Performance of K-Relaxed Variants

Takeaway: k-relaxed variants allow the user to gracefully explore the trade-off between invisibility and staleness, and enable more trade-off points that are not covered in base lenses

We evaluate the performance impact of k values in the k-relaxed variants. Recall that k value is the additional number of UCs the user allows to read the latest graph for the corresponding lenses: LCMB, LCNB, and GCNB. In this experiment, we vary the k from 0 to 22 with an interval of 2. Our results in Figure 16-17 show that the k-relaxed variants allow the user to gracefully explore the trade-off between invisibility and staleness, and enable more trade-off points that are not covered in base lenses. The results in Figure 16a show that as we relax more UCs, the invisibility increases for the k-relaxed variants. However, when k becomes the same as or larger than the viewport size (i.e., 4 in our test), the invisibility does not change for k-LCNB and k-LCMB since they have converged to GCPB. On the other hand, staleness decreases as we have a larger k as shown in Figure 16b.

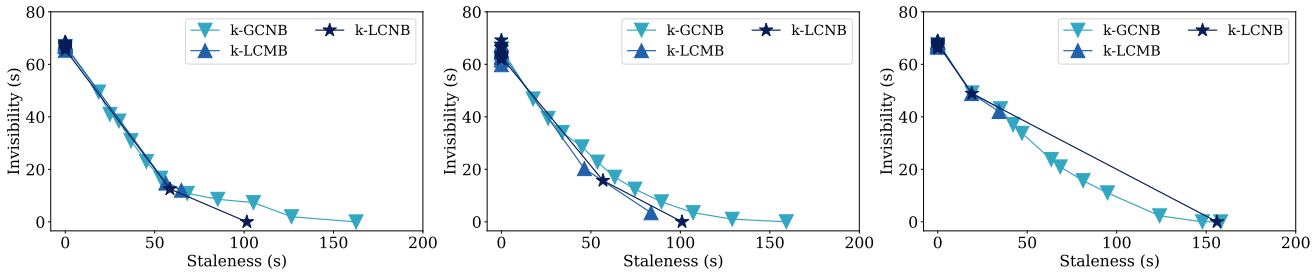
Figure 17 shows the trade-offs between invisibility and staleness under three read behaviors. We see that the k-relaxed variants have different trade-offs for different read behaviors. For example, for Regular Move in Figure 17a k-LCNB has better trade-offs than k-GCNB when the staleness is larger than 100 s, meaning that for the same invisibility, k-LCNB has smaller staleness than k-GCNB. When the staleness is smaller than 100 s, all of the three k-relaxed variants stay on the same trade-off curve. For Random Move, k-LCMB has the best trade-offs compared to the other two variants, and for Wait and Move, k-GCNB has the best trade-offs.

6.4 Effectiveness of Scheduler Optimizations

Takeaway: The optimized scheduler in transactional panorama reduces staleness and invisibility in most cases.

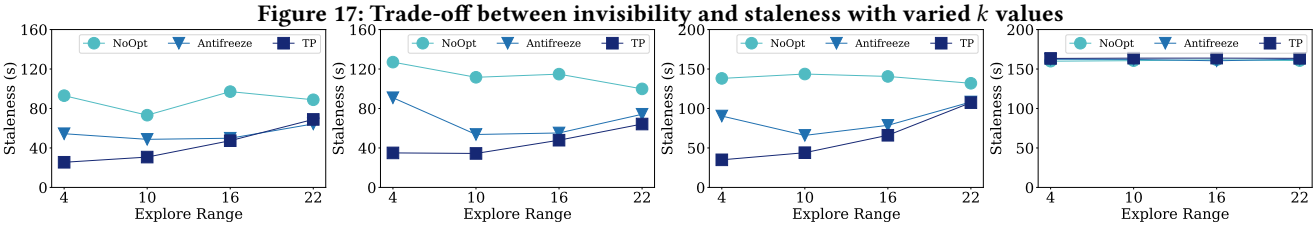
This experiment evaluates the performance benefit and overhead of the optimizations for the scheduler in Transactional Panorama (TP for short). We compare TP with two baselines: 1) NoOpt, after updating the base tables randomly picking a view to compute; 2) Antifreeze, an existing work that prioritizes computing the view with least execution time from a spreadsheet system [15]. Recall that in TP we use the metric in Section 4 to optimize the scheduler. It prioritizes computing the views whose query results are more likely to be read in the future and are faster to compute, where the former factor is estimated by the amount of time when a view was in the viewport. Therefore, the performance benefit of TP over the baselines is sensitive to the locality of reading the view graph: the likelihood of a view that was read in the past also being read in the future. To test the effect of locality, we vary the explore ranges and report the staleness and invisibility for different base lenss under different scheduling.

Figure 18 shows that TP has smaller staleness compared to NoOpt and Antifreeze for all of the lenses. The performance benefit of TP over the baselines is larger when we have smaller explore ranges.



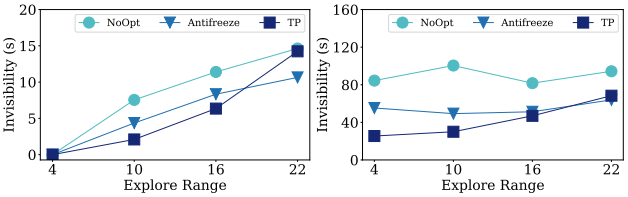
(a) Regular Move (b) Random Move (c) Wait and Move

Figure 17: Trade-off between invisibility and staleness with varied k values



(a) ICNB (b) LCMB (c) LCNB (d) GCNB

Figure 18: Evaluation of scheduler optimizations (staleness)



(a) LCMB (b) GCPB

Figure 19: Evaluation of scheduler optimizations (invisibility)

Specifically, TP reduces staleness by up to 75% and 62% compared to NoOpt and Antifreeze, respectively. TP and the baselines have the same staleness for GCNB because GCNB refreshes the views after all of the new results are computed and its staleness is independent of a scheduler policy. Figure 19 shows that TP has smaller invisibility compared to NoOpt and Antifreeze in most cases. Similar to the results of staleness, TP has a larger performance benefit when the explore range is smaller except the case of LCMB with explore range 4. In this case, the explore range equals to the viewport size, so LCMB does not have invisibility. Overall, TP reduces invisibility by up to 70% and 54% compared to NoOpt and Antifreeze, respectively. However, TP may have higher invisibility than Antifreeze when the locality of reading the view graph weakens, such as for LCMB with explore range being 22. In this case, TP increases invisibility by 33%.

7 RELATED WORK

transactional panorama is related to work in transaction processing, view maintenance and stream processing, and rendering of analysis results in visual interfaces.

Transaction processing. There is a long line of work on improving the performance of transaction processing while maintaining serializability or snapshot isolation [18, 19, 27, 31, 35, 37, 39]. For example, SNOW Theorem [27] studies fundamental trade-offs between the power (e.g., consistency level) and latency of read-only transactions, and defines the optimal properties read-only transactions can maintain simultaneously. On the other hand, SAGAS [23] focuses on breaking a long-lived transaction into sub-transactions, which can interleave with other concurrent transactions to improve performance. However, none of these projects consider maintaining consistency while reading uncommitted results or desired user semantics in visual interfaces, such as visibility and monotonicity. SafeHome [14] adapts transactions to define atomicity and serializability for concurrent routines in smart homes, and includes a series of visibility models that make trade-offs between performance and user visibility (i.e., what intermediate states of smart devices are visible to users). transactional panorama is different from SafeHome because we focus on an end-user data analysis scenario; so the MVC properties are not considered in SafeHome. The definition of "visibility" in SafeHome is also different ours.

View maintenance and stream processing. Many papers design incremental view maintenance algorithms to make view refresh more efficient [13, 20–22, 25, 32, 36, 40]. These techniques are orthogonal to our model and can be used to improve the performance of our system. S-Store [28] and transactional stream processing [17] integrate transactions into stream processing to guarantee consistency for shared states. In a related vein, Golab and Johnson [24] study different consistency levels for materialized views in a stream warehouse with respect to the source data, while Zhuge et al. [41] propose a notion, multiview, to define multiple views to be refreshed consistently. transactional panorama is different from these work because they do not consider the user's semantics of consuming the results in a visual interface along with properties, such as monotonicity and visibility.

Rendering analysis results in a visual interface. As summarized in Table 1, many existing data analysis tools, including Excel [6], Google Sheets [2], Dataspread [15], Libre Calc [5], Superset [11], Power BI [8], and Tableau [12] make fixed choices on the properties maintained while rendering analysis results with respect to an update. Interaction Snapshot [38] creates a scaled-down display of the dashboard for each interaction (e.g., cross-filtering) alongside, where the scaled-down dashboard serves as the new snapshot and uses an indicator to show whether the new snapshot

is computed. This way, the user can interact with the old snapshot and chooses to replace it with the new snapshot later, similar to GCNB. The difference is that Interaction Snapshot does not allow a user to read uncommitted results and choose the different properties they desire. Another line of research renders approximate results [29, 33] and refines them later, which is different from our scenario where the user needs to see exact results.

8 CONCLUSION

We introduce transactional panorama, a framework that studies the fundamental trade-offs among monotonicity, consistency, and visibility when a user examines query results in a visual interface under updates. By developing MVC Theorems, we identified feasible property combinations, some of which are novel. With the new performance metrics defined, we proved ordering relationships across the possible property combinations for the performance metrics. We additionally designed new algorithms for efficiently maintaining different property combinations and processing the updates. We implemented transactional panorama in a popular BI tool, Superset, and enabled the user to choose the properties and performance trade-offs they desire. Our experiments characterized the performance of different property combinations, showed significant performance difference across them, and demonstrated the values of our model and newly discovered property combinations.

REFERENCES

[1] Datadog. https://www.datadoghq.com/.
[2] Google sheets. https://www.google.com/sheets/about/.
[3] Grafana. https://en.wikipedia.org/wiki/Grafana.
[4] Kibana. https://en.wikipedia.org/wiki/Kibana.
[5] Libreoffice calc. https://www.libreoffice.org/discover/calc/.
[6] Microsoft excel. http://products.office.com/en-us/excel.
[7] Plotly. https://plotly.com/.
[8] Power bi. https://powerbi.microsoft.com/en-us/.
[9] Redash. https://redash.io/.
[10] Streamlit. https://streamlit.io/.
[11] Superset. https://superset.apache.org/.
[12] Tableau. https://www.tableau.com/.
[13] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.*, 5(10):968–979, 2012.
[14] S. B. Ahsan, R. Yang, S. A. Noghabi, and I. Gupta. Home, safehome: smart home reliability with visibility and atomicity. In A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 590–605. ACM, 2021.
[15] M. Bendre, T. Wattanawaroon, K. Mack, K. Chang, and A. G. Parameswaran. Antifreeze for large and complex spreadsheets: Asynchronous formula computation. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1277–1294. ACM, 2019.
[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
[17] I. Botan, P. M. Fischer, D. Kossmann, and N. Tatbul. Transactional stream processing. In E. A. Rundensteiner, V. Markl, I. Manolescu, S. Amer-Yahia, F. Naumann, and I. Ari, editors, *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 204–215. ACM, 2012.
[18] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, 2009.
[19] Y. Chen, X. Yu, P. Koutris, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu. Plor: General transactions with predictable, low tail latency. In Z. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 19–33. ACM, 2022.
[20] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.
[21] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 469–480, 1996.
[22] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 405–416, 1997.
[23] H. Garcia-Molina and K. Salem. Sagas. In U. Dayal and I. L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, pages 249–259. ACM Press, 1987.
[24] L. Golab and T. Johnson. Consistency in a stream warehouse. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 114–122. www.cidrdb.org, 2011.
[25] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 157–166, 1993.
[26] Z. Liu, B. Jiang, and J. Heer. *imMens*: Real-time visual querying of big data. *Comput. Graph. Forum*, 32(3):421–430, 2013.
[27] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The SNOW theorem and latency-optimal read-only transactions. In K. Keeton and T. Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 135–150. USENIX Association, 2016.
[28] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-store: Streaming meets transaction processing. *Proc. VLDB Endow.*, 8(13):2134–2145, 2015.
[29] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In G. Mark, S. R. Fussell, C. Lampe, m. c. schraefel, J. P. Hourcade, C. Appert, and D. Wigdor, editors, *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017*, pages 2904–2915. ACM, 2017.
[30] D. Moritz, B. Howe, and J. Heer. Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In S. A. Brewster, G. Fitzpatrick, A. L. Cox, and V. Kostakos, editors, *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, page 694. ACM, 2019.
[31] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689. ACM, 2015.
[32] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 511–526, 2016.
[33] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB*, volume 99, pages 709–720, 1999.
[34] A. S. Tanenbaum and M. van Steen. *Distributed systems - principles and paradigms, 2nd Edition.* Pearson Education, 2007.
[35] D. Tang, H. Jiang, and A. J. Elmore. Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* www.cidrdb.org, 2017.
[36] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Thrifty query execution via incrementability. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1241–1256. ACM, 2020.
[37] T. Wang, R. Johnson, A. D. Fekete, and I. Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.*, 26(4):537–562, 2017.
[38] Y. Wu, R. Chang, J. M. Hellerstein, and E. Wu. Facilitating exploration with interaction snapshots under high latency. In *31st IEEE Visualization Conference, IEEE VIS 2020 - Short Papers, Virtual Event, USA, October 25-30, 2020*, pages 136–140. IEEE, 2020.
[39] J. Zhang, K. Huang, T. Wang, and K. Lv. Skeena: Efficient and consistent cross-engine transactions. In Z. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 34–48. ACM, 2022.
[40] J. Zhou, P. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 231–242, 2007.
[41] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. Multiple view consistency for data warehousing. In W. A. Gray and P. Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, pages 289–300. IEEE Computer Society, 1997.