Visual Monitoring of Process Runs: An Application Study for Stored Procedures

Matthias Meyer Fabian Beck* Steffen Lohmann†
University of Stuttgart, Germany Fraunhofer IAIS, Germany

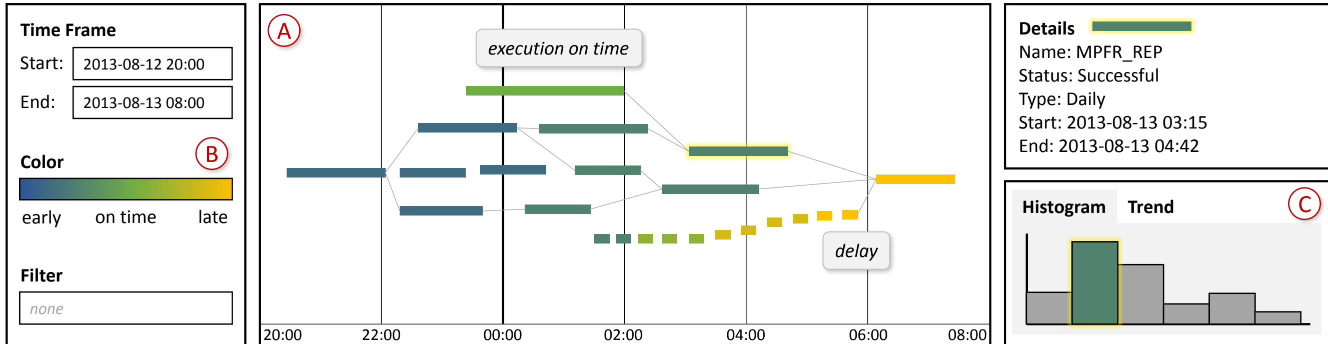


Figure 1: User interface sketch of the visualization approach representing the execution of stored procedures on a timeline (A). Deviations from other runs are encoded in color (B) and detailed in histogram and trend views (C).

Abstract

Stored procedures are used in database systems to process and aggregate data. Hundreds of stored procedures often form a complex process network with documented and hidden dependencies that is difficult to understand, maintain, and debug. This paper introduces a novel approach to support such tasks by visually comparing a specific process run to other runs of the same process. The visualization is based on a force-directed node-link diagram arranged on a timeline. Color coding, histograms, and trend charts are used to highlight temporal deviations. The approach has been implemented as an interactive web application and used by professional database developers for solving realistic maintenance and debugging tasks. The feedback of these expert users confirms the usefulness and practical relevance of the approach.

1 Introduction

Processes are monitored to warrant the correct behavior of a system. Experience helps the maintainers of the system judge whether a process they are observing runs within its usual boundaries or shows some uncommon behavior. For that, they do not have to remember every previous run of the process precisely. They rather develop a fuzzy *feeling* or *intuition* that tells them whether parts of the process behave in alignment to past runs. Deviations from historic patterns can be observed as subprocesses that take unusually long or as a changed order of a typical sequence of events.

Gaining sufficient experience to monitor a process like this, however, takes time, and hence, is costly. Moreover, if the process exceeds a certain size and complexity, it becomes difficult if not impossible to develop the required intuition. Subjective judgment or change blindness could further bias perception. This paper investigates how visualization can support users to monitor processes in an objective and reliable way by comparing them to previous runs. The goal is to aid inexperienced as well as experienced observers to detect anomalies and problems in a specific run of a process.

The focused application are stored procedures used in database and data warehouse systems to process and aggregate data. The stored procedures run as a sequence of connected subprocesses that are repeatedly executed according to a schedule with explicitly or implicitly defined dependencies. The execution times change depending on the data contained in the database and the general workload of the servers. Since hundreds of stored procedures might need to be executed and, typically, dependencies are only partly documented, even small deviations in the execution schedule could lead to performance loss or errors. The database administrators and developers need to locate the problem and understand its cause.

Our visualization approach to support maintainers of database systems in this regard is illustrated in Figure 1: Its main view (A) is based on a node-link diagram depicting an individual execution of the stored procedures. Each node represents an executed procedure and is arranged on a timeline according to its start and end time. Documented dependencies are shown as links. An adapted force-directed layout algorithm arranges the nodes in a vertical order. Deviations from previous runs are encoded by color according to a color scale (B). The user can retrieve details on demand that provide more context for temporal deviations in the form of histograms and trend charts (C). The example given in Figure 1 shows a run of the stored procedures that starts on time, even a little earlier than usual, until a sequence of shorter events at the bottom of the node-link diagram causes a delay and leads to a late termination.

We see our main contributions in mapping a relevant database maintenance problem to a visualization question (Section 3) and presenting an interactive solution (Section 4). The approach is novel, as previous work has not investigated this application or, in general, has not studied deviations of process executions from time schedules or historic runs in sufficient detail (Section 2). Having worked closely together with an industry partner and tested the tool in a realistic scenario with domain experts gives us confidence that our approach provides a valuable tool for database administrators and developers (Section 5).

*e-mail: fabian.beck@visus.uni-stuttgart.de
†e-mail: steffen.lohmann@iais.fraunhofer.de

2 RELATED WORK

Visualizing stored procedures as a process network on a timeline means applying graph visualization techniques to a software engineering problem. Although the procedures are executed in a specific temporal order, we do not focus on *dynamic graph visualization* as defined by Beck et al. [5], which would require analyzing a changing graph topology over time. Our visualization approach is related to *visual graph comparison* (e.g., [1, 4, 18, 28]). However, instead of comparing multiple graphs, we study a specific batch run of stored procedures as a graph in detail and only integrate temporal context from other runs by color coding and details on demand.

Software systems are often visually abstracted as graphs, representing not only static but also dynamic dependencies recorded at runtime of the systems. UML sequence diagrams [8] are a standardized way to encode object activity and dynamic dependencies: objects are visualized as linked bars on a vertical timeline. Although being intended to be a modeling tool, these diagrams can be also applied to visualize recorded interaction data of objects [20] or web services [14]. Zinsight [11, 12] applies a similar technique to operating system events, organizing them by time, type, and project space in a sequence-diagram-like main view; additional views can be used to retrieve color- and space-encoded timing information. Greevy et al. [23] extend a sequence diagram into 3D to visually stack instances of the same classes. Our visualization approach also uses rectangles to indicate activity, but does not assign each stored procedure an individual column or row because too many procedures need to be visualized.

Holten et al. [24] focus on the scalable visualization of dynamic dependencies by drawing massive sequences of links between classes organized in columns, without depicting activity periods on the timeline. With a similar focus, Beck et al. [6] visualize dynamic calls as a sequence of changing graphs in juxtaposed columns. However, these approaches only explore one execution while we study multiple repeated ones.

Another abstraction of execution information are stack traces, which are often depicted as hierarchies of executed methods in forms of icicle plots [13]; these, however, only work well for synchronous calls, but could not easily encode asynchronous executions like we have for stored procedures. Other visualizations depict different tasks or processes on a horizontal timeline, for instance, as rows connected by links [25, 29] or as bundled bars [15] (more examples are surveyed by Isaacs et al. [26]). In this context, however, we only found one approach that discerns and visualizes multiple executions: Trümper et al. [33] compare two executions by juxtaposing their hierarchical stack trace visualizations and connecting similar execution phases by bundled links.

Our approach can also be classified as a visualization technique of time-oriented data [2]. It is related to project planning diagrams like Gantt charts [10, 19] and diagrams used in the modeling of algorithms, workflows, and business processes (e.g., flow charts, BPMN, EPCs). Planning Lines [3] extend Gantt charts to encode temporal uncertainty: rectangles representing the activities are adapted showing, for instance, earliest and latest starting times. TASM [32] compares multiple versions of schedules based on Gantt charts and network layouts. Although these approaches provide possibilities to compare several instances of a process, they are not targeted at contrasting a selected instance of a process to hundreds of previous instances as required in our scenario. LiveGantt [27] demonstrates that Gantt charts can be made scalable; they, however, do not show dependencies between activities.

Most visualizations of time-oriented data (e.g., Gantt charts) can be used to visually compare several instances by juxtaposing or overlaying diagrams [22]. While these visual comparison approaches allow to easily compare a few instances, we want to contrast a single selected instance with a long history of previous instances. This can be described as an asymmetric visual comparison [7]. For this purpose, an explicit encoding of the difference as described in the taxonomy of Gleicher et al. [22] offers a solution: we encode time differences explicitly in the color of graph nodes. While this approach scales to comparing a current to many historic instances, it is only an indirect, aggregated form of comparison.

In general, we are not aware of any approach that contrasts a selected process run to other runs of the same system to analyze anomalies in the temporal behavior of the selected one, neither for stored procedures nor for comparable application scenarios. Moreover, supporting database developers and maintainers monitoring their systems of stored procedure or other data transition infrastructures seems to be an underexplored area of research. Although we focus on a specific area of application, related areas such as dynamic graph visualization, visualization of software behavior, and visualization of time-oriented data could profit from our results and the general lessons learned.

3 VISUALIZATION PROBLEM

The first step of building a visualization approach that supports database administrators and developers is to transform the domain problem into a visualization problem. To this end, we first analyze the specific application background, then clarify terminology and build a graph-based data model for stored procedures, and finally formulate requirements for the visualization approach.

3.1 Application Background

Stored procedures are used in relational databases and data warehouse systems to transform and aggregate data. They can be considered as small programs that assemble multiple SQL queries, contain conditional expressions, loop constructs, and allow the declaration of variables [17]. Stored procedures are supported by all major relational database management systems, though following slightly different standards. A typical application scenario of stored procedures is the following: data is collected in a production database system and batch processed stored procedures are used to transform and aggregate the collected data into a data warehouse system. During batch execution, the production system, however, has to be inactive, which is the case for many systems at night. Advantages of this architecture are two decoupled database systems that both work with high performance after the data update is completed.

For this work, we collaborated with a medium-sized wholesale company. Their main data warehouse system, used by about 400 end users, serves as our application example throughout the paper. For this system, they use the described batch processing architecture to update the data. Different batch executions are applied for daily, weekly, monthly, quarterly, semiannual, and annual updates, which are triggered according to a specific schedule (i.e., always at the same time of the respective day). Stored procedures follow a strict naming scheme that encodes the type and purpose of the procedure, distinguishing between aggregation, export, data loading, data transformation, and service procedures. As a basis for comparison, we use all executed batch runs of the stored procedure system from January 1, 2000 to September 4, 2013. Within those more than thirteen years, 2,808 different stored procedures were executed, totaling up to more than 3.5 million individual executions of procedures. The database developers confirmed that the system of stored procedures, the database software, and server hardware were only changed gradually in this frame of time. The only exception was a major hardware upgrade that happened at the end of the studied period—speed up effects were expected.

Within a batch run, the stored procedures need to be executed in a specific order. Otherwise, it is no longer warranted that the results are a valid aggregation of the data because many procedures contain implicit assumptions on preprocessing steps. These assumptions are documented as predecessor–successor dependencies and stored

in a separate database. The developers try to document all dependencies and keep this documentation up to date. However, it is difficult to maintain a complete and consistent record of all dependencies. Further, some dependencies are too complex to be mapped to the database scheme. In consequence, there also exist undocumented dependencies that cannot be explicitly retrieved from the database; they are only implicitly defined through the batch routines of the system.

3.2 Terminology and Data Model

In this paper, we visualize the batch executions of a system of stored procedures, in short *executions* or *runs*. We use the terms *stored procedure* and *procedure* interchangeably. An *executed procedure* denotes an executed instances of a stored procedure being part of a batch run. The naming scheme used in our application example allows us to identify semantically related procedures, called *similar procedures*: procedures sharing at least three of five critical literals in their names (with few domain-specific exceptions).

Formally, we model a set of batch executions of a stored procedures system as a directed graph $G = (V, E)$ where a node $v \in V$ represents an executed instance of stored procedure $sp(v)$. We define the set of all executed instances of a specific procedure p as $V(p) = \{v \in V | sp(v) = p\}$. Each executed procedure v is assigned a *start time* $t_{\text{start}}(v)$ and an *end time* $t_{\text{end}}(v)$ where $t_{\text{start}}(v) < t_{\text{end}}(v)$; the *runtime* of the executed procedure is defined as $\Delta_t(v) = t_{\text{end}}(v) - t_{\text{start}}(v)$. Further, the *reference time* $t_{\text{ref}}(v)$ is the earliest timestamp of the data that is processed as part of the batch execution that v is part of (e.g., if data of interval $[t_1, t_2]$ is processed in a batch execution, $t_{\text{ref}}(v) = t_1$ for all executed procedures v of the batch execution); we use it to define a *relative start time* $rt_{\text{start}}(v) = t_{\text{start}}(v) - t_{\text{ref}}(v)$. A documented predecessor–successor dependency between two executed procedures v_1 and v_2 is modeled as a directed edge $(v_1, v_2) \in V$; v_1 is called *predecessor* of v_2 and v_2 is called *successor* of v_1.

To contrast an executed procedure v with all executed instances $V(sp(v))$ of the procedure $sp(v)$, we compute the median $\overline{rt}_{\text{start}}(v)$ of all relative start times $rt_{\text{start}}(v')$ over all instances of the procedure $v' \in V(sp(v))$ (since the dataset often contains skewed distributions and extreme outliers, we do not use mean values). The *lateness* of v is the deviation of the relative start time $rt_{\text{start}}(v)$ from this median: $l(v) = rt_{\text{start}}(v) - \overline{rt}_{\text{start}}(v)$. Hence, a positive lateness value indicates a later relative execution than in the median case, and a negative value an earlier one. Analogously, we calculate the median runtime $\overline{\Delta}_t(v)$ over all runtimes $\Delta_t(v')$ with $v' \in V(sp(v))$ and define the *runtime deviation* of v as $rd(v) = \Delta_t(v) - \overline{\Delta}_t(v)$. A positive runtime deviation, hence, means a runtime longer than median, a negative value a shorter one.

3.3 Requirements

As part of the *discover phase* [30] at the beginning of the project, we discussed the application in detail with our industry partner. One of the authors had considerable experience in the domain because he had worked for the industry partner as a database developer. He led the discussions and acted as an intermediary between domain and visualization experts. Target users of the approach are the database maintainers and developers, who monitor and extend the described data warehouse system on a daily basis.

From those discussions, we learned that the system of stored procedures is difficult to maintain because it consists of a high number of procedures and contains complex, partly undocumented dependencies. The developers would appreciate tool support for maintenance tasks of the system, such as monitoring the behavior of the system, optimizing its performance, finding sources of errors or decays of runtime speed. Observing effects of gradual changes in the data, stored procedures, or hardware onto the behavior and schedule of the system are further relevant use cases. Since these tasks are rather exploratory, a visualization approach will likely provide the necessary flexibility.

Thus, the visualization approach has to show a system of executed stored procedures as defined in the data model. Since the developers and maintainers usually focus on the analysis of a specific run, the tool does not need to show all executions at the same time, but it is sufficient to show time frames of only a few hours up to a few days. Still, the history of executions should be easily explorable by switching the time frame. Documented and undocumented dependencies structure the executed procedures. For each executed procedure, it is further important to provide data from all other runs of this particular procedure. In summary, the specific requirements of the visualization tool are to

1. give an **overview of the temporal sequence of procedures** of a selected execution,
2. reveal and depict **documented and undocumented dependencies** between executed procedures, and
3. provide **temporal context** for the selected execution to show deviations from the schedule observed in other executions.

To address these requirements in a visualization approach, a *design* and *implementation phase* [30] followed the *discover phase*: We applied an iterative process, discussing a design decision, implementing it in a prototype, collecting feedback from industry stakeholders, and continuing with a refinement or other design decisions. The following section describes the outcomes of this process.

4 Visualization Approach

Our visualization approach is illustrated in Figure 1, while a screenshot of its implementation as a tool is shown in Figure 2. The graph of executed procedures is arranged as a node-link diagram on a timeline in the main view (Requirement 1). A specialized force-directed layout algorithm arranges the nodes vertically, placing nodes nearby that are linked or share similar properties according to domain-specific definitions (Requirement 2). While the diagram shows only one run of the system, context from the history of runs is provided by coloring the nodes according to their lateness (or other measures, such as runtime or status) and showing details in histograms and trend diagrams (Requirement 3). Interactions further ease the exploration of the data with our approach.

We implemented the visualization approach as a web-based tool, using JavaScript with the visualization library D3 [9] for the front end and, on the server side, Python with Flask. The server application wraps the database access through a REST web service providing the data in JSON format for the front end.

4.1 Graph Layout

We visualize the graph of executed procedures as a 2D node-link diagram because this type of diagram is easy to interpret and allows for a flexible arrangement of the nodes on a timeline. In contrast, an adjacency matrix representation might be more scalable [21] but is harder to explain to users, is more difficult to combine with a timeline, and would be mostly empty because the graph of stored procedures is usually quite sparse. Also, path-related tasks (e.g., identifying chains of executed procedures) are important, which are difficult to perform in matrix representations [21].

In the following, we discuss the layout of the diagram and the representation of nodes. While it is straightforward to visualize documented dependencies, a particular challenge is to reveal undocumented ones. They are not detected automatically in our approach because one needs additional domain knowledge to finally decide. However, there exist indicators that two executed procedures depend on each other: first, the potential successor is executed soon after the potential predecessor ended; second, they share a similar lateness; and third, they are similar regarding the scheme

Figure 2: Visualization tool showing a nightly run of a stored procedure system and showing details of a selected procedure DWT1010P executed on August 1, 2013; the respective node is highlighted in the figure by a red rectangle.

encoded in their name (*name similarity*). Therefore, we place nodes vertically close to each other if they are similar or have similar lateness (*lateness similarity*).

Timeline: Since an overview of time is our first requirement, we use a timeline as the horizontal axis of the node-link diagram. Each executed procedure $v \in V$ is represented by a rectangular node, having its left side aligned with the start time of the executed procedure $t_{\text{start}}(v)$, and its right side with the end time of the executed procedure $t_{\text{end}}(v)$. Hence, similar as in a Gantt chart, the width of the nodes scales according to the runtime of the represented executed procedure $\Delta_t(v)$. The time constraints dictate the horizontal arrangements of nodes, but we are still free to move the nodes vertically. To find a good vertical arrangement, we implemented a layout algorithm that is described below. Edges in the graph representing the documented dependencies are drawn as straight links. Although these edges are directed, we do not need to draw arrow heads or use other encodings of direction because the context of the timeline already clarifies the predecessor–successor direction of dependence.

Graph Simplification: First experiments of visualizing the graph on a timeline already showed us that a frequent pattern appears and dominates the visualizations: single executed procedures depending on a set of procedures previously executed in parallel having no dependencies to previous nodes. In the visualization, these form fan-like structures [16] because multiple links merge into a single node. While these patterns produce considerable clutter in the visualization, their information density is rather low: they

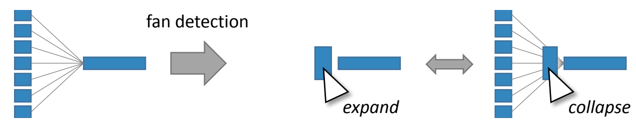


Figure 3: Graphical simplification of a fan pattern in the node-link diagram, illustrating the interactive collapsing and expanding of a subgraph implementing the fan pattern.

typically represent simple data loading processes. Inspired by the graph motif simplification of Dunne and Shneiderman [16], our approach detects these fan patterns automatically and visualizes them in an simplified form as illustrated in Figure 3: all predecessor nodes in the fan are collapsed into a thin but higher rectangle of a fixed size closely attached to the successor node.

Formally, we collapse those groups of at least five nodes (a) that do not have any predecessors, (b) that have the same successors, and (c) whose successor does not have any other predecessors not being part of the group. In contrast, the successor, which represents the merging point of the fan, is preserved. This simplification largely reduces the visual clutter produced by a fan pattern, without destroying its characteristic triangular shape. Clicking on the collapsed node expands the fan again to its original representation. To circumvent a layout update of the whole graph whenever a fan node

is expanded, enough space is reserved by default, since the layout algorithm described in the following considers expanded fans.

Vertical Arrangement of Nodes: While the horizontal position of the stored procedures is predetermined by the timeline, we need an appropriate layout to vertically arrange the nodes. It soon became clear that a layout like in Gantt charts, where each node is represented in a separate row, would be too space-consuming because of the high number of procedures that we have to deal with. Instead, we decided to arrange the nodes freely on the vertical axis, optimizing the following criteria:

1. avoid overlap of nodes,
2. reduce link lengths to place dependent nodes close,
3. keep the graph compact,
4. spatially separate isolated nodes (i.e., nodes without any dependencies) from those having incoming or outgoing dependencies (isolated nodes are moved to the top),
5. place nodes with a high *lateness similarity* close to each other to make undocumented dependencies visible,
6. place nodes being part of the group of predecessors in fans close to their successor, and
7. place nodes with a high *name similarity* close to each other to indicate semantic groups.

Force-directed graph algorithms allow for encoding layout optimization goals like these as forces. However, they usually arrange nodes in a 2D layout while, in our case, the horizontal position of the node is determined already by the timeline. To apply a force-directed algorithm, hence, we first have to restrict the arrangement of nodes in the algorithm to vertical movement.

Our implementation is based on the force-directed layout of the D3.js library [9]. By default, this algorithm supports optimization criteria 1-3. To limit the algorithm to vertical positioning, we set the horizontal component of the nodes according to their position on the timeline and skip all changes of the horizontal component in the force-directed algorithm. To implement optimization criteria 4-7, we added an extra force for each of them. These forces contain parameters we tuned until the algorithm produced satisfying results.

Since one of our main goals was to reveal undocumented dependencies, the results might not be optimal considering only compactness (3). While undocumented dependencies are not explicitely computed, they become explorable through placing nodes with a similar lateness next to each other (5). Also, semantic grouping (7) improves to build semantic relationships between nodes that have no documented dependencies. These extensions produce virtual lines of implicitly connected nodes in the layout.

4.2 Temporal Context

Laying out the graph of executed procedures as described above provides a good overview of a single run of the system. However, when we do not only want to rely on experience, anomalies in the execution only become visible if we add temporal context to the diagram that summarizes the temporal behavior of all runs. For each current instance of a procedure, this context can be summarized by distributions and deviations of time. While aggregated time deviations are indicated by color in our approach, time distributions are visualized with histograms and trend diagrams.

Color Coding: To make the temporal context graspable at a quick glance, we encode it as an attribute of the nodes in the node-link diagram. Color provides a simple, yet powerful means to implement such an encoding of a single attribute. Individual deviations from all runs, as discussed in Section 3.2, are represented by the lateness or runtime deviations of the executed procedures. We encode these two metrics in the color of the nodes. In the tool, the coloring scheme can be switched between the two metrics; a third color coding shows status information as a categorical encoding.

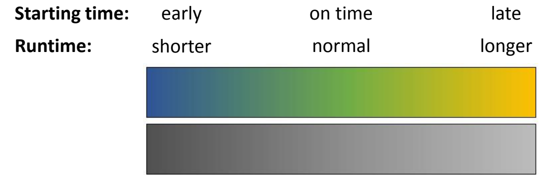


Figure 4: Color scale used for encoding the lateness and runtime deviations of executed procedures (top); transformed to a gray scale, a brightness gradient is revealed (bottom).

For mapping the lateness (or runtime deviation) to color, we tested several color scales. Requirements were that the scale is intuitive to read, provides enough resolution to discern values, and is still readable for people with common color vision deficiencies, such as red-green color blindness. We finally decided to use a color scale from blue to amber as illustrated in Figure 4, top:

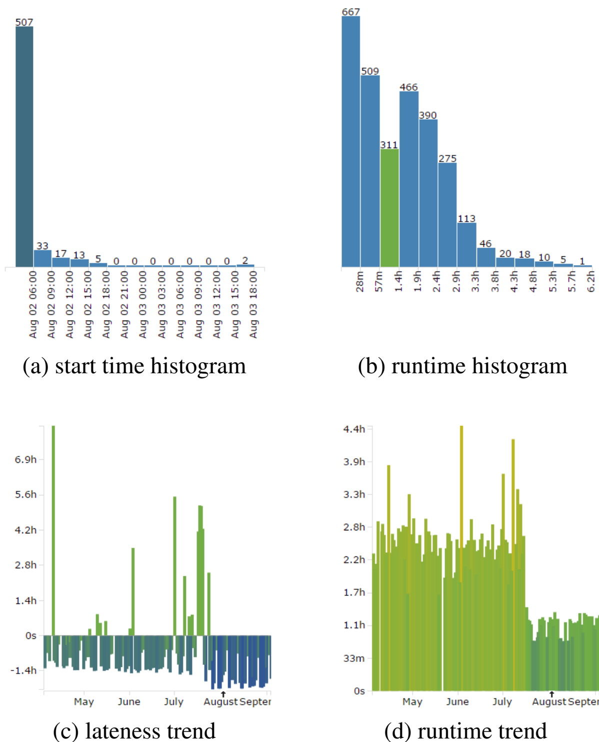
- **blue**, attributed as a neutral and calm color, encodes procedures executed earlier (shorter) than the median;
- **green**, attributed with balance and harmony, encodes procedures started at (ran for) comparable time as the median;
- **amber**, usually used as a warning color, encodes procedures run later (longer) than the median.

When transforming the color scale to a gray scale as shown in Figure 4, bottom—thereby simulating what a fully color-blind user sees—we observe a slight brightness gradient. This gradient does not just warrant the color scale to be readable without seeing color, but increases the perceivable resolution of the color scale as well. We also tested other color vision deficiencies using the Coblis color blindness simulator and confirmed the suitability of the color scale. The color scale can be considered as a mix of a sequential scale (sequential luminance gradient) and a diverging scale (diverging hue using three base colors).

Like Figure 2 shows for lateness values, it is easy to spot late procedures using this color scale. Color also visually groups procedures, because elements of similar color are perceived as similar according to Gestalt laws [34]. This implicit grouping is also motivated from a domain perspective: procedures of similar lateness are likely to be connected by implicit, undocumented dependencies, which now become apparent due to color coding. Together with the layout arranging nodes close that have a high *name* and *lateness similarity*, clearly discernible strings of nodes are formed that make hidden dependencies visible (e.g., Figure 2, top).

Histograms and Trend Charts: Providing a single color-coded value for each stored procedure gives a good overview; however, it does not show a complete picture of how the current execution deviates from other runs. In addition to just comparing the current execution to median values, seeing the distribution of lateness values and runtime or the evolution of those variables would certainly provide extra value. Yet, integrating this information into the graph view is difficult and would overload the diagram. Hence, we added this information to an extra view, which can be displayed on demand. It shows the detailed information summarizing all other instances $V(sp(v))$ of a selected procedure v.

In this view, we discern the four different bar charts shown in Figure 5. Histograms aggregate all executions into bins to show the distribution of the start times (Figure 5, a) and of runtime values (Figure 5, b). In both diagrams, the bar highlighted in a different color represents the bin that contains the current execution. The two trend charts use a timeline as vertical axis to show the evolution of



(a) start time histogram

(b) runtime histogram

(c) lateness trend

(d) runtime trend

Figure 5: Histograms and trend charts relate a selected executed procedure to all instances of the procedure based on start time, runtime, and lateness; like in Figure 2, procedure DWT1010P executed on August 1, 2013 is selected.

lateness values (Figure 5, c) and runtime values (Figure 5, d). A small arrow on the timeline marks the current execution. We applied the same coloring scheme as for coloring the nodes representing the executed procedures (cf. Section 4.2)

4.3 Interactive Interface

The tool embeds the described visualization approach into an interactive interface. In a navigation pane (Figure 2, top left), users select the time frame and types of stored procedures (i.e., daily, weekly, monthly, etc.). The side panel on the left further provides options to switch the color scale and to filter the stored procedures. Standard panning and zooming interactions are implemented for the main view. It is possible to select chains (i.e., paths) starting and ending in a particular execution; all elements are faded out in this case, except those that are connected transitively to the selected procedure. Details for individual stored procedures are retrievable on demand. The histograms and trend charts are part of those details, but a textual pane provides more details, such as name, status, type, and the precise start and end time of the executed procedure (Figure 2, top right). Finally, a log of events as written out by the database server is provided. The log contains important information such as statistics on the processed data or error messages. The log view shows just the events related to an individual executed procedure or also the context of the full sequence of events.

Loading a typical 24h interval of the described dataset takes 7–8s on a consumer PC (Intel i7-3517U, 4GB RAM, SSD, Windows 7, Chrome 47, Oracle XE 11g). The force-directed algorithm takes about another 5–7s to produce a stable layout. After that, interactions like pan, zoom, and select run smoothly without notable lags.

4.4 Example

To give an example of an analysis, we point out some observations in Figure 2 and Figure 5. The visualizations show data from the nightly batch run of August 1, 2013. As can be observed in the node-link view of Figure 2, a cluster of densely connected procedures forms in the middle. Like expected due to data loading processes, fan patterns are detected and aggregated mainly in the first third of the timeline. One fan-like pattern is not collapsed—it does not form a pure fan because it has one predecessor also connected to other nodes. A set of procedures that are likely to be connected by undocumented dependencies produces a long sequence of blue boxes in the top part of the view.

Previously, around middle of July, a major server update was installed, significantly increasing the server performance. As an expected consequence, blue colors dominate the graph view, indicating that the executed procedures ran earlier than the median case. Selecting one of the executed procedures that consumed reasonable runtime, like DWT1010P in Figure 2, we see the sudden drop of runtime in the chart in the right side panel. The lateness trend for the same selected executed procedure in Figure 5 (c) confirms an earlier execution after the upgrade; the improvement here, however, is much smaller than with respect to runtime. We also observe occasional delays before the upgrade that vanished after. The histograms in Figure 5 (a) and (b) further reveal that the start time of the selected procedure is early (within a very skewed distribution) and that the runtime is shorter than usual, but it had already been much shorter for other runs. Going back to Figure 2, we also find that the upgrade did not bring a major advantage in all areas: many green and even some nearly amber nodes indicate procedures that ran as late or even later than the median. The upgrade, hence, only had partial success.

5 Qualitative Evaluation

We conducted an expert study to evaluate the usefulness of our approach. The study took place in a meeting room at the wholesale company we collaborated with (cf. Section 3.1). Seven database experts participated in the study and independently solved realistic analysis and debugging tasks. They had each worked for 5 to 20 years with database systems and are responsible for the development, maintenance, and customization of the stored procedures infrastructure in the company. In particular, they are in charge of changing and configuring the stored procedures, tables, and dependencies, and resolving issues that occur during execution.

5.1 Procedure

The tool was installed on a laptop and shown on a 24” monitor with a resolution of 1920x1080 pixels. It was executed in a web browser (Chrome) and connected to the database of the company to access the latest data. This included the latest log files of the stored procedures that were loaded into the tool right before the study started. The tool was started with an initial time period for the visualization, comprising a nightly run from 8 p.m. to 8 a.m. of the next day. While comments concerning the usability of the tool were appreciated, we asked the expert users to focus their feedback on the general visualization approach.

The expert users took part one at a time. The study started with an introduction into the visualization approach, followed by an explanation of the graphical user interface and interactive features of the tool. The participants could then make themselves familiar in a free exploration phase, and ask any questions required to fully understand the visualization approach and its implementation. Subsequently, they had to solve three tasks with the tool:

1. Which procedure was running the second-longest on day X?
2. When was the last successor of procedure Y completed in the batch run of day X?

3. What was the reason why procedure *Y* did not end as usual at 9 a.m. on day *X*?

(with each *X* being a given day in the past and each *Y* being the name of a stored procedure, such as DWT1030P).

The three tasks are common examples of what the expert users have to deal with in their daily routine. Stored procedures with a long runtime are often a candidate for improvement, which is also the case for overly long chains of stored procedures. In particular, Task 1 is an example of checking the most critical procedures, which could indicate a general performance issues (debugging scenario) or a problem of an individual execution (monitoring scenario). Task 2 checks if participants could follow dependencies with the help of the visualization; in practical monitoring application, this kind of task answers whether the processing of a specific dataset (identified by an initial loading procedure) has already terminated. Task 3 is very common as part of an error analysis in stored procedures. Although timing did allow us to investigate only a very limited set of tasks, our selection covers a variety of realistic scenarios including both monitoring and debugging.

After the completion of each task, the participants had to answer the following evaluation questions:

1. Is the visualization approach helpful in solving the task?
2. Is it easier or harder than before to solve the task? Why?
3. What functionality is missing to properly solve the task?

With these questions, we wanted to examine whether the visualization approach is an improvement over existing solutions. However, participants were allowed to also use their common work environment in the study consisting of tools such as Quest Toad for Oracle, Oracle SQL Developer, Token 2, and FileZilla.

After completing the three given tasks, the expert users were asked to choose a fourth task by themselves that they would like to solve with the tool. Lastly, they had to answer four final questions that evaluate the general applicability as well as benefits and limitations of the approach:

1. For what kind of tasks could the approach be useful?
2. Which expectations were not met by the approach?
3. What worked well, what could be improved?
4. Is the visualization of the daily batch run useful?

While the study targets at a realistic setting involving the actual maintainers of a database system, the resulting study design also comes along with some limitations: First, the number of participants is low and does not allow a quantitative evaluation of the results. Second, different biases might have influenced the results: first, the developers were very accustomed to a specific set of tools they had used every day over years and might be reluctant to use any new tools and change their workflow. Second, the participants had known and worked together with one author of this paper before, which could have affected their objectivity. Third, the limited time the participants could invest for the study restricted the number, complexity, and realism of tasks we could test.

5.2 Results

Overall, the visualization approach received very positive feedback. Six of the seven participants considered the visualization helpful in solving the tasks. All but one stated that the tasks were generally easier to solve with the visualization than they would be without. This result was strongest for the second task where all seven users agreed that the visualization helped a lot, while most considered it very useful for Task 3 and quite useful for Task 1.

Task Completion: The lower rating for Task 1 is little surprising, as it is difficult to precisely compare the length of bars that are not aligned, i.e., have varying start and end points [31]. For that reason, the expert users could have solved Task 1 more efficiently with a simple SQL statement than with the visualization. However, the visualization provides context information, such as the dependencies between stored procedures, which could be useful for follow-up tasks. Some expert users therefore proposed to extend the visualization by further filters that allow to show, for instance, only the *x* longest procedures.

The experts agreed that Task 2 was well supported by the visualization of the stored procedure chain. They could easily spot and follow the chains in the visualization, which was further facilitated through interactively highlighting selected chains by fading out the rest. Some participants noted that chains can split and run in parallel, which could make it hard to determine which procedure ends last. Although the vertical grid lines in the background support the visual comparison, the study participants were sometimes not sure and looked up the exact end times in the details view.

Task 3 required to identify a large delay between two consecutive procedures, which was not a problem for any of the study participants. They stressed the fact that the visualization facilitates this kind of tasks, as delays in chains are clearly indicated by overly long edges. Some users concluded from the log that an external system might be the reason for the delay. However, all users stated that they would require additional information to verify these and other assumptions, and some proposed to include further data sources in the visualization, especially dependencies to external systems.

Free Exploration and Discussion: During the free exploration, the expert users tended to analyze procedures in the center of the visualization as well as those developed by themselves. The latter were merely used to check whether the visualization is correct and displays the procedure and dependencies as expected. Likewise, most participants chose a debugging case as fourth task, in which they either focused on a single procedure and its dependencies or on the complete graph structure of the batch run. For instance, one user analyzed the batch run of the day when the study took place, and found that, against expectations, some procedures have not yet been started. However, he also noted that it is difficult to say for sure which procedures should have been started, as only a part of the dependencies are explicitly documented.

Most participants stated that they would use the tool to analyze the structure of the stored procedure system and see where it could be improved. While error analysis was mentioned as a possible application area, the expert users stressed the limitations of the approach with regard to finding the reasons for errors. Two participants mentioned that they could imagine using the visualization to document stored procedures and adapt the schedule. All users but one liked the overview of the batch run that is provided by the visualization. Some criticized the visual clutter in the overview, but added that it gets less problematic as soon as one analyzes the visualization in more detail using the zoom and other interactions.

The experts reported several features useful for a more comprehensive analysis. Repeatedly mentioned was the inclusion of alternative views on the stored procedures, especially a sortable table listing different attributes of the stored procedures, such as runtime or lateness. Related to this, features were proposed to filter the stored procedure visualization based on such attributes. The expert users noted that some of these filters may already be applied before the data is loaded into the visualization tool to reduce the amount of data that needs to be transferred between server and client. Finally, comments concerned the inclusion of further data sources, such as dependencies to external systems or additional log files.

6 Conclusions

We have introduced an approach for the visual monitoring of process runs that supports a number of maintenance tasks by visual comparison of a specific process run to other runs of the same pro-

cess. We have demonstrated the usefulness of the approach by applying it to a large database containing log data recorded over a time period of more than thirteen years. Scalability is achieved by restricting the analysis to a certain time frame and providing various options for filtering and interaction. We adapted the force-directed layout in a way that it results in a less cluttered node-link diagram.

The feedback of the expert users was very positive and confirms the usefulness of the approach. Despite some observed visual clutter, the overview visualization was considered valuable to get a better impression of the actual schedule and dependencies of the stored procedures as well as the temporal deviations. Furthermore, the analysis of chains of stored procedures was highly supported by the visualization according to the expert users. This includes the discovery of errors and unexpected system behavior, such as the identification and analysis of procedures that run delayed or crashed.

As part of future work, we want to generalize the approach to related applications, for instance, executions of general software systems, loading behavior of web pages, or executed instances of business and production processes. Further, we are interested to extend the approach towards comparing specific instances of runs, categorizing runs, and detecting outliers within the set of runs. Also, the force-directed algorithm could be further optimized, for instance, by increasing the compactness of the representation.

References

[1] A. Abuthawabeh, F. Beck, D. Zeckzer, and S. Diehl. Finding structures in multi-type code couplings with node-link and matrix visualizations. In *Proceedings of the 1st IEEE Working Conference on Software Visualization (VISSOFT '13)*, pages 1–10. IEEE, 2013.

[2] W. Aigner, S. Miksch, H. Schumann, and C. Tominski. *Visualization of time-oriented data*. Springer, 2011.

[3] W. Aigner, S. Miksch, B. Thurnher, and S. Biffl. PlanningLines: novel glyphs for representing temporal uncertainties and their evaluation. In *Proceedings of the 9th International Conference on Information Visualisation (InfoVis '05)*, pages 457–463. IEEE, 2005.

[4] K. Andrews, M. Wohlfahrt, and G. Wurzinger. Visual graph comparison. In *Proceedings of the 13th Conference on Information Visualisation (IV '09)*, pages 62–67. IEEE, 2009.

[5] F. Beck, M. Burch, S. Diehl, and D. Weiskopf. A taxonomy and survey of dynamic graph visualization. *Computer Graphics Forum*, 2016, to appear.

[6] F. Beck, M. Burch, C. Vehlow, S. Diehl, and D. Weiskopf. Rapid serial visual presentation in dynamic graph visualization. In *Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '12)*, pages 185–192. IEEE, 2012.

[7] F. Beck, F.-J. Wiszniewsky, M. Burch, S. Diehl, and D. Weiskopf. Asymmetric visual hierarchy comparison with nested icicle plots. In *Joint Proceedings of the 4th International Workshop on Euler Diagrams and the 1st International Workshop on Graph Visualization in Practice (GraphVIP '14)*, volume 1244 of *CEUR-WS*, pages 53–62, 2014.

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley Professional, 2nd edition, 2005.

[9] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[10] W. Clark, W. N. Polakov, and F. W. Trabold. *The Gantt Chart: A working tool of management*. Ronald Press Company, 1922.

[11] W. De Pauw and S. Heisig. Visual and algorithmic tooling for system trace analysis: a case study. *ACM SIGOPS Operating Systems Review*, 44(1):97–102, 2010.

[12] W. De Pauw and S. Heisig. Zinsight: A visual and analytic environment for exploring large event traces. In *Proceedings of the 5th International Symposium on Software Visualization (SoftVis '10)*, pages 143–152. ACM, 2010.

[13] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Software Visualization*, volume 2269 of *LNCS*, pages 151–162. Springer, 2002.

[14] W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. F. Morar. Web Services Navigator: visualizing the execution of web services. *IBM Systems Journal*, 44(4):821–845, 2005.

[15] W. De Pauw, J. L. Wolf, and A. Balmin. Visualizing jobs with shared resources in distributed environments. In *Proceedings of the 1st IEEE Working Conference on Software Visualization (VISSOFT '13)*, pages 1–10. IEEE, 2013.

[16] C. Dunne and B. Shneiderman. Motif simplification: improving network visualization readability with fan, connector, and clique glyphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*, pages 3247–3256. ACM, 2013.

[17] A. Eisenberg. New standard for stored procedures in SQL. *ACM SIGMOD Record*, 25(4):81–88, 1996.

[18] M. Freire, C. Plaisant, B. Shneiderman, and J. Golbeck. ManyNets: An interface for multiple network analysis and visualization. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems (CHI '10)*, pages 213–222. ACM, 2010.

[19] H. L. Gantt. *Work, Wages, and Profits*. Engineering Magazine Company, 2 edition, 1913.

[20] P. Gestwicki and B. Jayaraman. Methodology and architecture of JIVE. In *Proceedings of the 2005 ACM Symposium on Software Visualization (SoftVis '05)*, pages 95–104. ACM, 2005.

[21] M. Ghoniem, J.-D. Fekete, and P. Castagliola. On the readability of graphs using node-link and matrix-based representations: A controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005.

[22] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4):289–309, 2011.

[23] O. Greevy, M. Lanza, and C. Wysseier. Visualizing live software systems in 3D. In *Proceedings of the 2006 ACM Symposium on Software Visualization (SoftVis '06)*, pages 47–56. ACM, 2006.

[24] D. Holten, B. Cornelissen, and J. J. Van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '07)*, pages 47–54. IEEE, 2007.

[25] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann. Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2349–2358, 2014.

[26] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the art of performance visualization. In *EuroVis - STARs*, pages 141–160. Eurographics Association, 2014.

[27] J. Jo, J. Huh, J. Park, B. Kim, and J. Seo. LiveGantt: Interactively visualizing a large manufacturing schedule. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2329–2338, 2014.

[28] R. Lutz and S. Diehl. Using visual dataflow programming for interactive model comparison. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*, pages 653–664. ACM, 2014.

[29] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[30] M. Sedlmair, M. Meyer, and T. Munzner. Design study methodology: Reflections from the trenches and the stacks. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2431–2440, 2012.

[31] J. Talbot, V. Setlur, and A. Anand. Four experiments on the perception of bar charts. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2152–2160, 2014.

[32] M. Tory, S. Staub-French, D. Huang, Y.-L. Chang, C. Swindells, and R. Pottinger. Comparative visualization of construction schedules. *Automation in Construction*, 29:68–82, 2013.

[33] J. Trümper, J. Döllner, and A. Telea. Multiscale visual comparison of execution traces. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC '13)*, pages 53–62. IEEE, 2013.

[34] M. Wertheimer. Untersuchungen zur Lehre von der Gestalt. II. *Psychologische Forschung*, 4(1):301–350, 1923.