Exploratory Development of Data-intensive Applications

Sampling and Streaming of Large Data Sets in Live Programming Environments

Patrick Rein
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Marcel Taeumel
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
marcel.taeumel@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

Michael Perscheid
SAP Innovation Center
Potsdam, Germany
michael.perscheid@sap.com

ABSTRACT

Business applications are usually data-intensive. The process of designing and implementing such applications benefits from working with realistic data to sharpen requirements and discover pitfalls. However, such data is usually quite extensive and the feedback cycles during programming and design activities can become long and distracting. As a result, programmers might prefer abstract thinking and mental simulations over working with concrete, realistic data. We propose a new approach supporting live programming, with immediate feedback and explorable runtime data, for the domain of data-intensive business applications on top of relational databases. With the integration of streamed access to sampled data, we can employ productive traits of a live programming environment such as Squeak/Smalltalk, which is not optimized for the processing of huge amounts of data and is hence not well-suited for such tasks. We describe two representative scenarios and also discuss limitations by putting our approach in relation to the current development of business applications.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments**; • **Information systems** → Middleware for databases;

KEYWORDS

live programming, business applicationd evelopment, streaming, sampling, relational databases

ACM Reference format:
Patrick Rein, Marcel Taeumel, Robert Hirschfeld, and Michael Perscheid. 2017. Exploratory Development of Data-intensive Applications. In *Proceedings of Programming Experience Workshop, Brussels, Belgium, April 03, 2017 (PX/17),* 11 pages.
DOI: http://dx.doi.org/10.1145/3079368.3079399

1 INTRODUCTION

Live programming – editing a program while it is running – promises short feedback loops for developers in situations where requirements and constraints are not well-known in advance to development. This exploratory style of programming is well-supported by various environments for different use cases, such as developing graphical desktop applications, developing web applications, and even live music performances [6, 9, 10, 13, 27]. However, for applications working with large data sets, such a live programming experience is difficult to provide because live programming partly relies on a sense of immediacy, which is supported by tools for manipulating runtime data and a short timespan between changes to source code and perceiving changed behavior in the system [21, 26]. Computation on large data sets requires considerable time and thus delays feedback, as a visible change in the behavior of the system takes longer to emerge. Further, as data sets are large and often stored in remote systems, such as relational databases, interactively exploring or manipulating the data is often cumbersome as, for example, data has to be loaded from the remote system first and often requires a manual conversion of data types. Beyond that, most application development environments are not designed for developers using concrete query results directly to build an application from it.

For the use case of analyzing large data sets, several systems have been designed to address this challenge, for example Sintr [3] or Tempe [4]. Both systems are designed for a scenario in which developers interact with the system to analyze a given data set. Large data sets, however, do not only occur in the context of data analysis but also in the context of common application development.

An example of a business application which is based on computations on large data sets, is the implementation of the dunning process in companies. The dunning process is similar to an analytical scenario as it requires the aggregation of past billing and shipping data for all customers [18]. From the perspective of a user however, the application for viewing overdue customers and sending out dunning letters is transactional. Thus, even when developing applications such as tools for the dunning process, developers face similar issues regarding response times and manipulation capabilities for runtime data as they face in the context of data analysis.

In the end, this also impedes the feedback cycle between customers and developers. A short delay between a modification of source code and a visible change in the behavior of the application could support participatory design [23], as customers can directly provide feedback on changes.

Thus, we propose a programming environment which brings the idea of live programming, with immediate feedback and explorable runtime data, to the domain of developing data-intensive business applications on top of relational databases. It provides two tools: an SQL workspace for creating queries and a table view for viewing result sets. The tools are build on two key mechanisms: streaming and sampling. The first provides timely feedback by streaming the results of long-running queries in order to provide developers with early feedback even if the final result is not known yet. The second mechanism enables the immediate exploration and manipulation of data-sets through providing samples of the underlying data set instead of the full data set. Thereby, developers can experiment with the concrete query results without using up resources of the development environment. Further, the data is provided as first class objects of the surrounding development environment and can thus be inspected with the same tools which can be used to inspect common runtime objects. We have implemented a first prototype of these tools in Squeak/Smalltalk and used them to create two exemplary business applications. While one is a generic histogram tool which can be used for any table, the other is a real-world use case around the dunning process in companies.

Overall our contributions in this paper are:

- The design of tools (SQL workspace and table view) and means (sampling and streaming) for improving feedback in the development of data-intensive business applications working with relational databases
- A first prototypical implementation of the tools and mechanisms in Squeak/Smalltalk [9]
- Two exemplary scenarios for developing graphical business applications in which our proposed live programming environment for data-intensive applications can be beneficial: A tool supporting dunning as a business process, and a generic histogram for exploring query results

The remainder of the paper is structured as follows. In Section 2, we give a short overview of live programming concepts and systems based on immediate feedback and explorable runtime data. In section 3 we will briefly outline the mechanisms and their implementations for enabling live programming for application development with large data sets. The workflow for developing graphical business applications in our environment is described in in Section 4 based on two example scenarios. In Section 5, we compare our approach to existing environments for business application development before we conclude the paper and give an overview of our next steps in Section 6.

2 LIVE PROGRAMMING AND BUSINESS APPLICATION DEVELOPMENT

Before describing our approach, we first give an overview of the nature of business application development and how it could generally benefit from a shorter feedback loop. We then introduce the aspects of immediacy and tangibility of live programming environments as a possible approach to shortening the feedback loop in business application development. We also describe the Squeak/Smalltalk system, which already covers many of described aspects of live programming.

2.1 Business Application Development

Short feedback cycles can be beneficial for the development of business applications. While there are standardized product lines for enterprise resource planning (ERP) systems, many components are still developed for individual companies and their particular business processes [20]. For the success of an ERP system deployment it is essential, that the business processes implicit in the ERP system fit the company's processes [8]. If an adaptation of the business processes within the company is not desirable, the ERP system is customized. To bring the customizations of the ERP system close to the customer's processes, developers sometimes work directly at the location of the customer. Ideally, they also directly involve the people affected by the process to be implemented [23].

Further, working directly with existing data of the customer's ERP system data is crucial for the successful development of these applications. Using generated test data is often not viable or not useful as it can not reflect the particularities of actual production data in an ERP system. For example, if the database structure has been modified over a long period of time by various parts of the organization, the structure does often not match the standard structure anymore. Further, the values used in certain fields can have special meaning in the context of the one organization. For example, it might be customary for users of a form for entering insurance claims to enter 99999 in the zip code field, if the zip code is unknown.

Generally speaking, when working directly with customer data and at the customer's location, timely feedback in such a scenario could enable developers to directly get feedback from the customer. Thereby, they could determine whether the application does fulfill the requirements and could enable them to directly incorporate the customer feedback even during discussions.

2.2 Liveness and Immediate Feedback

One way to enable this short feedback loop is to enable programmers to directly edit the application while it is running. Thereby, they can evolve the code directly based on the observed behavior of the application working on concrete data. In general, the feature of editing an application while it is running is at the core of live programming environments.

To bridge the gap between the static representation of the application behavior and its actual dynamic behavior, many live systems provide quick feedback on changes to the behavior of the system as a result of a change in the source code [17, 26]. Thereby, they strive for an experience of enabling the developers "to edit a program while it is running" [25]. While there are no explicit recommendations on upper bounds of the response time of the system the system should generally continue execution "without noticeable interruption according to the updated version of the program" [25]. A noticeable interruption occurs, for example, if a query is send to a database, and the completion of the query takes several seconds. Besides that, the context in which the change was applied should be preserved, for example graphical objects should not be deleted and re-created but continue to be visible, now behavior according to their new behavior [7].

2.3 Liveness and Tangible Objects

In order to support the exploration of the dynamic behavior of the system further, some environments provide means to inspect, navigate, and manipulate runtime data [6, 26]. These means make runtime data tangible, as developers can directly interact with them. The impression of tangibility can be amplified by providing some kind of visual representation which can be used to interact with the object, for example a generic tool which displays the attributes of an object or a list of objects which can be manipulated through context menus. In contrast, in systems in which objects are not tangible, developers might only be able to read values of certain objects they specified at compile time.

Prior work hints, that making exemplary data accessible in the development environment can also be beneficial for the co-creation of software by programmers and non-programmers [11]. Thus, it could also be beneficial for the context of business application development when business domain experts and programmers develop software together.

At the same time, the visual representation of runtime objects does not have to constitute a complete direct manipulation interface which requires the representation of objects to lend to some degree to physical metaphors [21]. Arbitrary runtime data might be too abstract for such a representation to exist.

Overall, through tangible objects developers can interactively explore the dynamic relationships of objects which result from the static source code.

2.4 Squeak/Smalltalk

Squeak/Smalltalk is a live programming environment which can be used to build graphical desktop applications. As it is based on the Smalltalk language, everything in the system is an object, including the meta-structures of classes and methods [6]. Squeak/Smalltalk support the hot-swapping of code by replacing the method object in the method dictionary of the class object. As the compilation and hot-swapping take place in under 50 ms the application can continue without a “noticeable interruption” [19].

Also, most of the environment is self-supporting as the development tools, such as the code browser, the debugger, or the inspector for inspecting the state of objects, are all written in Smalltalk and are running in the same process [6, 9]. Squeak/Smalltalk makes runtime data tangible to some extent through the object inspector tool (see Figure 3 for a Smalltalk-style inspector). It allows developers to inspect and navigate through all objects available in the system. By executing small scripts on the object, the inspector also allows for manipulating these objects.

While live programming environments, such as Squeak/Smalltalk, are well-suited for general application development, the liveness can lead to challenges regarding large data-sets. For example, as Squeak/Smalltalk is self-supporting, development tools and application data share the same resources. Thus, a large data set uses up memory which is also required by the environment itself. Similarly, when processing large amounts of data within the environment, the processing time is not available for the interactive tools and thus the overall environment might become unresponsive.

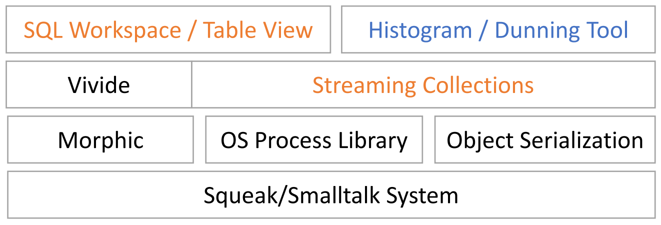


Figure 1: A diagram illustrating the basic architecture of our approach. Our contributions are highlighted in orange and the tools created in the case studies are highlighted in blue. All four are created using the Vivide tool building framework and use the streaming collection data structure.

3 MEANS FOR LIVE PROGRAMMING OF BUSINESS APPLICATIONS

We will describe the tools and mechanisms designed for providing a live programming experience. The tools are based upon the Vivide tool-building environment [24] which is an essential part of the resulting environment. Figure 1 shows the basic relation between the mechanisms we propose, the created development and application tools, and the surrounding libraries and frameworks.

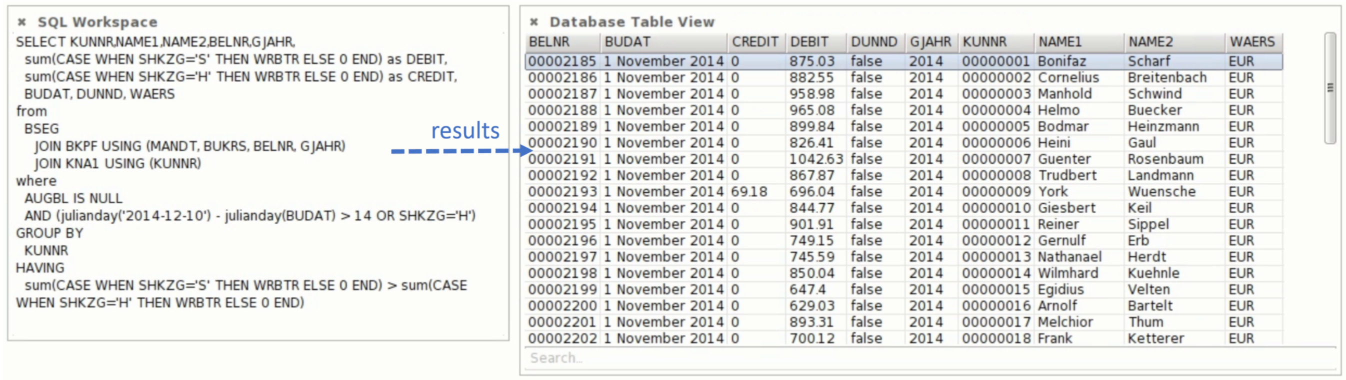
3.1 Tools

The main entry point for developers of data-intensive business applications in our environment are two tools: the SQL Workspace and the Table View. To illustrate the difference between their functionality and the functionality provided by the surrounding Vivide environment we first give a short introduction to Vivide.

3.1.1 The Vivide Environment. Vivide is an extension of the Squeak/Smalltalk environment which eases the development of graphical tools by taking a data-driven perspective on them [24].

As a framework, it helps developers to develop graphical tools by separating the code which prepares the data to be displayed from the code which deals with the details of graphical components, like rendering or event handling. As a programming environment, it changes the workflow, as the objects themselves become central to user interactions. For example, instead of opening a class browser, the developer opens a set of class objects, optionally filters and transforms them, and finally selects a graphical component to display the resulting data set. It also allows the programmer to combine existing tools by defining a data flow between them, such as the connection between the input sandbox and the parse result visualizer.

Scripts and Views. In Vivide, the code which prepares the data for displaying is represented as a *script*. The visual component displaying this data is called a *view* and can be a list, a tree, or even a three dimensional visualization of a graph. Typically, one tool has several scripts which transform domain objects to the target data set and the corresponding view configurations. A view will get the resulting view configuration as an input and render the elements appropriately. For example, if the collection of methods is rendered as a list, then each method becomes a list item with the



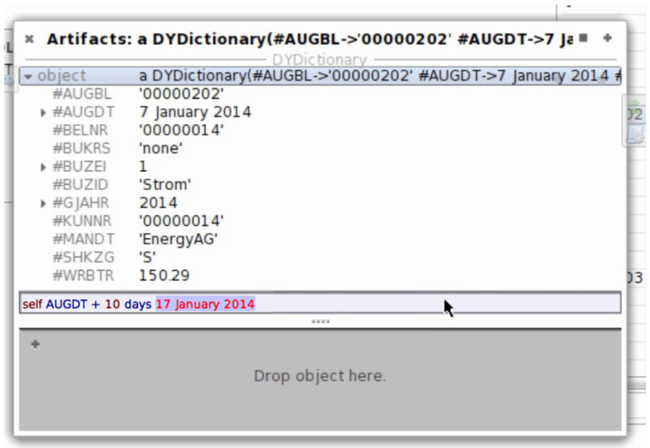
```
SELECT KUNNR,NAME1,NAME2,BELNR,GJAHR,
  sum(CASE WHEN SHKZG='S' THEN WRBTR ELSE 0 END) as DEBIT,
  sum(CASE WHEN SHKZG='H' THEN WRBTR ELSE 0 END) as CREDIT,
  BUDAT, DUNND, WAERS
from
  BSEG
    JOIN BKPF USING (MANDT, BUKRS, BELNR, GJAHR)
    JOIN KNA1 USING (KUNNR)
where
  AUGBL IS NULL
  AND (julianday('2014-12-10') - julianday(BUDAT) > 14 OR SHKZG='H')
GROUP BY
  KUNNR
HAVING
  sum(CASE WHEN SHKZG='S' THEN WRBTR ELSE 0 END) > sum(CASE
WHEN SHKZG='H' THEN WRBTR ELSE 0 END)
```

BELNR	BUDAT	CREDIT	DEBIT	DUNND	GJAHR	KUNNR	NAME1	NAME2	WAERS
00002185	1 November 2014	0	875.03	false	2014	00000001	Bonifaz	Scharf	EUR
00002186	1 November 2014	0	882.55	false	2014	00000002	Cornelius	Breitenbach	EUR
00002187	1 November 2014	0	958.98	false	2014	00000003	Manhold	Schwind	EUR
00002188	1 November 2014	0	965.08	false	2014	00000004	Helmo	Buecker	EUR
00002189	1 November 2014	0	899.84	false	2014	00000005	Bodmar	Heinzmann	EUR
00002190	1 November 2014	0	826.41	false	2014	00000006	Heini	Gaul	EUR
00002191	1 November 2014	0	1042.63	false	2014	00000007	Guenter	Rosenbaum	EUR
00002192	1 November 2014	0	867.87	false	2014	00000008	Trudbert	Landmann	EUR
00002193	1 November 2014	69.18	696.04	false	2014	00000009	York	Wuensche	EUR
00002194	1 November 2014	0	844.77	false	2014	00000010	Giesbert	Keil	EUR
00002195	1 November 2014	0	901.91	false	2014	00000011	Reiner	Sippel	EUR
00002196	1 November 2014	0	749.15	false	2014	00000012	Gernulf	Erb	EUR
00002197	1 November 2014	0	745.59	false	2014	00000013	Nathanael	Herdt	EUR
00002198	1 November 2014	0	850.04	false	2014	00000014	Wilmhard	Kuehnle	EUR
00002199	1 November 2014	0	647.4	false	2014	00000015	Egidius	Velten	EUR
00002200	1 November 2014	0	629.03	false	2014	00000016	Arnolf	Bartelt	EUR
00002201	1 November 2014	0	893.31	false	2014	00000017	Melchior	Thum	EUR
00002202	1 November 2014	0	700.12	false	2014	00000018	Frank	Ketterer	EUR

Figure 2: A screenshot of the SQL workspace on the left and the TableView on the right. The two windows are connected so that the result data of the SQL query flows from the SQL workspace to the TableView.

method selector as its label. The combination of scripts and a view is captured in a *pane*, for example the rule browser combining a script for extracting rules from a grammar and a list view.

Data-driven Tool Construction. Every pane has an output which is determined by the view. For example, when a user selects a method from a list view containing methods, the output of the pane of the list is the selected method object. This output can be connected to the input of another pane. For example, a developer could create a list of classes and connect its output to the input of our pane, listing the methods of a class. These combinations of panes through connections can also be grouped together to form single tools.



object	a DYDictionary(#AUGBL->'00000202' #AUGDT->7 January 2014 #
#AUGBL	'00000202'
#AUGDT	7 January 2014
#BELNR	'00000014'
#BUKRS	'none'
#BUZEI	1
#BUZID	'Strom'
#GJAHR	2014
#KUNNR	'00000014'
#MANDT	'EnergyAG'
#SHKZG	'S'
#WRBTR	150.29

```
self AUGDT + 10 days 17 January 2014
```

Figure 3: A Vivide object inspector showing the contents of a data record. The input field below the list allows for the execution of scripts on the object.

3.1.2 SQL Workspace. The SQL workspace (see Figure 2) can be used to edit SQL queries and send them to a database. Developers can edit the query and on hitting the saving keyboard shortcut the query is send off to the database. The workspace produces a query result object. This object can directly be used to create new tools by using it as example input for a Vivide script. Thereby, it is an entry point to developing a new application.

The concrete database that the query is sent to, can be defined globally for the environment or locally for a single workspace. As the workspace is a Vivide pane it provides the query result object at its outgoing data flow port. Whenever, the query is saved, the workspace produces a new result object at its outgoing port.

3.1.3 Table View. The database table view (see Figure 2) displays query results originating from the SQL workspace. It displays the result as a table, as it resembles the model behind relational databases. At the same time, each row in the table is an actual object. It can be dragged out of the list and when dropped on the screen, the environment opens an object inspector for it (see Figure 3) which can be used to inspect and manipulate the record. Changes to the record object are not propagated back to the database (see Section 3.2.3). The table view itself is also a Vivide pane and provides the currently selected rows at its outgoing port.

3.2 Mechanisms

3.2.1 General Architecture. The goal of our approach is to enable interactive programming. Hence, we have to keep the user interface of the programming environment responsive even when developers create long-running queries. Therefore, we generally separate the graphical user interface from the processing of queries. To implement this, we create a new operating-system-level process of the virtual machine and let this second process handle the communication with the database and the conversion of objects. Thereby, our graphical development environment is not affected by the processing and memory demands of the database communication. To get the query results, we create a proxy object in the development environment which represents the results of the query. This proxy regularly polls the remote process to fetch successfully processed data. Also, the proxy provides a collection interface which allows developers to use it like a common Smalltalk collection.

3.2.2 Sampling for Making Large Data Sets Tangible. A major issue for working interactively with large data-sets is their size. It is often not possible to get an exact understanding of the whole data set. Further, exploring a large data set through scripting in the development environment is infeasible due to resulting long processing times. If the data is stored in a database, the problem is worsened as developers can only start working with the data after the transmission of the result set. We propose to mitigate this issue by developing applications by only using samples of the data. The basic idea of this mechanism is that developers only work with samples of the original query result in the development environment. These samples should ideally contain a much smaller but nevertheless representative sample of the result. This sample can then be used to inspect the data and to develop the application.

Further, to still provide insights on the overall structure of the data set, the object representing the query result also provides methods to access value distributions of the single columns of the result and the overall result count.

Implementation. The sampling mechanism can be implemented either as part of the SQL query or in the environment itself. When implementing it by extending the original database query, one would wrap the SQL query in another query which only returns a subset using the `LIMIT` operator or the `SELECT TOP` clause. This approach will, however, only return the first results of the query and not a real random sample. Alternatively, the sampling could be implemented by sorting the query results using a random number and limiting the resulting results. This approach suffers from the constraint, that the complete query result has to be computed first to provide an ordering which can then be used to return the sample. Further, even the random ordering approach does not provide a representative sampling. However, even a stored procedure which produces representative samples would require a complete query result. A heuristic approach would be needed to provide both, short query execution time and representative results.

As our design favors quick feedback on query results, we have implemented the simple case of limiting the query results. Also, the query result object provides methods to get the value distribution of any field so developers can get an impression of the actual values which could occur.

3.2.3 Making Query Results Tangible.

Squeak/Smalltalk provides tools for inspecting and manipulating runtime objects. At the same time, when working with database data, the data is often only available as numbers or strings. To be able to use the tool set with query results, we return the result records as objects. Therefore, we first map primitive values to actual Smalltalk objects, for example values representing a Date to `Date` objects. We then create an object which has getter methods for all fields of the record and fill the object with the fields of the record. Generating generic setter methods is not trivial, as the query could have generated a projection which makes it impossible to map a field in the result to a field in a database table.

Implementation. The conversion of data values to corresponding Smalltalk objects is partially done by the database access library. However, for example in the dunning scenario (see Section 4), the data is mostly stored in strings. To still obtain objects, we have implemented simple heuristics which infer the type of the field from the syntax of the string representing it. After the values are all converted, they are stored in a `DYDictionary`, which maps the fields names to its values. To enable access to the fields via message sends, the `DYDictionary` class overrides the `doesNotUnderstand:` callback which is called whenever a message was not understood. The callback then checks the message name and if it matches a key in the dictionary returns the value. The complete conversion is executed in the database communication process in order to not use up resources of the interactive environment process.

3.2.4 Streaming into Collections for Providing Immediate Feedback.

Returning only a subset of the overall query results shortens the time between sending a query and getting first results. However, when the query requires a lot of computation per result record, even fetching the limited sample might take a long time and thereby inhibit the sense of liveness in the system. Thus, our environment streams all data which is retrieved from the database in the way Sintr and Tempe demonstrated [3, 4]. This includes, overall query results, samples, as well as information on the distribution of values in columns. The focus is on reducing response times for individual queries. Thus, the streaming stops as soon as the entire information was retrieved. In particular, it does not update the query result when the database changes.

As query results should be tangible, the streaming is hidden from the application developer. The environment provides a collection interface on the query result, as mentioned before, and the collection content is continuously updated in the background. To deal with these changes in the content of the collection, the result collection additionally provides an observer mechanism which notifies clients whenever the collection changed.

Implementation. As described before the retrieval and conversion of data from the database happens in a separate process. Such a process is started for each query and for each request for the value distribution of a column. After retrieving and converting the data, these processes write the result objects onto a `ReferenceStream` which converts the objects into a transmission format. This data is then written to an operating system pipe. In the development environment a periodic callback is executed which reads the other end of the pipe through a `ReferenceStream` and adds them to the corresponding result collection.

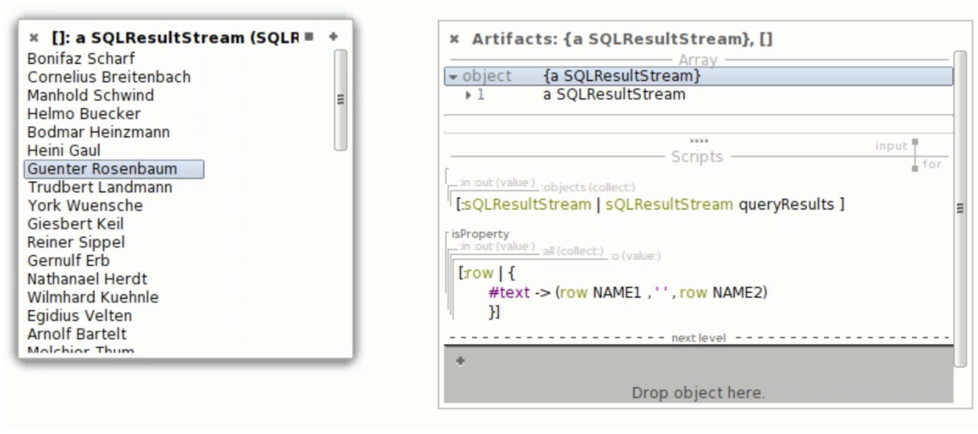
The collection integrates with the Vivide notification system. Whenever, the collection has been updated the collection sends out an event. Tools can register for this event from that particular collection.

4 BUILDING DATA-INTENSIVE APPLICATIONS IN A LIVE ENVIRONMENT

Our environment can be used to develop business applications and data analysis tools. We demonstrate both use cases. First, we show how to incrementally build a graphical tool supporting the dunning process based upon a real-world business use case. Second, we demonstrate that the same mechanisms can also be used to build general data analysis tools, such as a histogram.

4.1 Building a Dunning Application

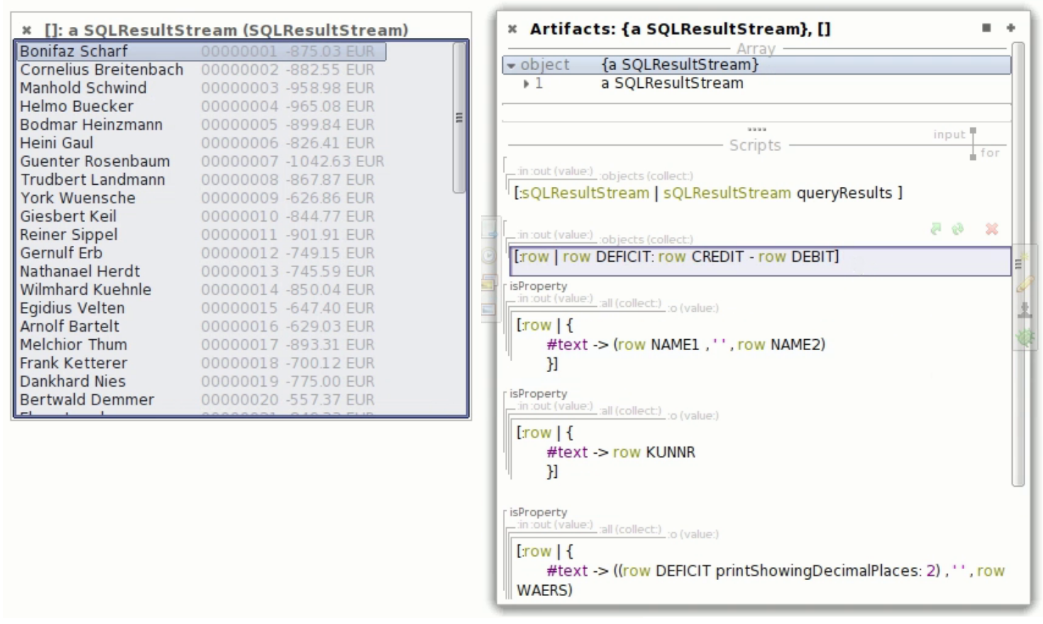
The dunning process is an important financial process in ERP systems. The analytical part determines the set of customers with open debit. Some ERP systems allow for automatic sending of dunning notices [5]. In our scenario, we will develop a graphical tool



```
[]: a SQLResultStream (SQLR
Bonifaz Scharf
Cornelius Breitenbach
Manhold Schwind
Helmo Buecker
Bodmar Heinzmann
Heini Gaul
Guenter Rosenbaum
Trudbert Landmann
York Wuensche
Giesbert Keil
Reiner Sippel
Gernulf Erb
Nathanael Herdt
Wilmhard Kuehnle
Egidius Velten
Arnolf Bartelt
```

```
Artifacts: {a SQLResultStream}, []
Array
object      {a SQLResultStream}
  1         a SQLResultStream

Scripts
:in :out (value:) :objects (collect:)
[:sQLResultStream | sQLResultStream queryResults ]

isProperty
:in :out (value:) :all (collect:) :o (value:)
[:row | {
    #text -> (row NAME1 , ' ' , row NAME2)
    }]

next level
Drop object here.
```

(a) First version of dunning tool

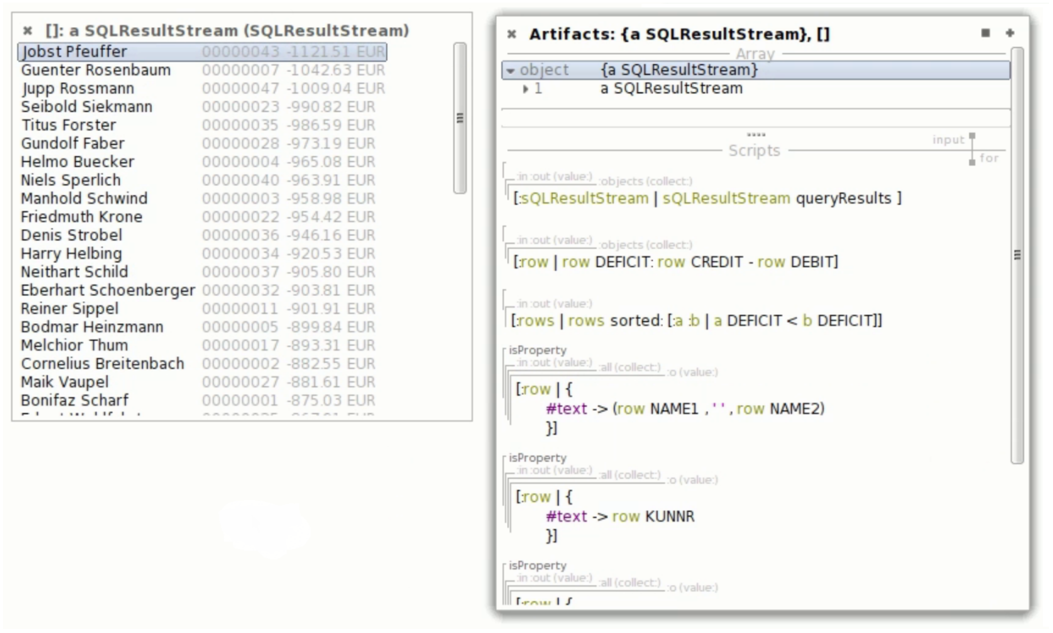


```
[]: a SQLResultStream (SQLResultStream)
Bonifaz Scharf          00000001 -875.03 EUR
Cornelius Breitenbach   00000002 -882.55 EUR
Manhold Schwind         00000003 -958.98 EUR
Helmo Buecker           00000004 -965.08 EUR
Bodmar Heinzmann        00000005 -899.84 EUR
Heini Gaul              00000006 -826.41 EUR
Guenter Rosenbaum       00000007 -1042.63 EUR
Trudbert Landmann       00000008 -867.87 EUR
York Wuensche           00000009 -626.86 EUR
Giesbert Keil           00000010 -844.77 EUR
Reiner Sippel           00000011 -901.91 EUR
Gernulf Erb             00000012 -749.15 EUR
Nathanael Herdt         00000013 -745.59 EUR
Wilmhard Kuehnle        00000014 -850.04 EUR
Egidius Velten          00000015 -647.40 EUR
Arnolf Bartelt          00000016 -629.03 EUR
Melchior Thum           00000017 -893.31 EUR
Frank Ketterer          00000018 -700.12 EUR
Dankhard Nies           00000019 -775.00 EUR
Bertwald Demmer         00000020 -557.37 EUR
```

```
Artifacts: {a SQLResultStream}, []
Array
object      {a SQLResultStream}
  1         a SQLResultStream

Scripts
:in :out (value:) :objects (collect:)
[:sQLResultStream | sQLResultStream queryResults ]

:in :out (value:) :objects (collect:)
[:row | row DEFICIT: row CREDIT - row DEBIT]

isProperty
:in :out (value:) :all (collect:) :o (value:)
[:row | {
    #text -> (row NAME1 , ' ' , row NAME2)
    }]

isProperty
:in :out (value:) :all (collect:) :o (value:)
[:row | {
    #text -> row KUNNR
    }]

isProperty
:in :out (value:) :all (collect:) :o (value:)
[:row | {
    #text -> ((row DEFICIT printShowingDecimalPlaces: 2) , ' ' , row
WAERS)
```

(b) Adding the deficit of each customer

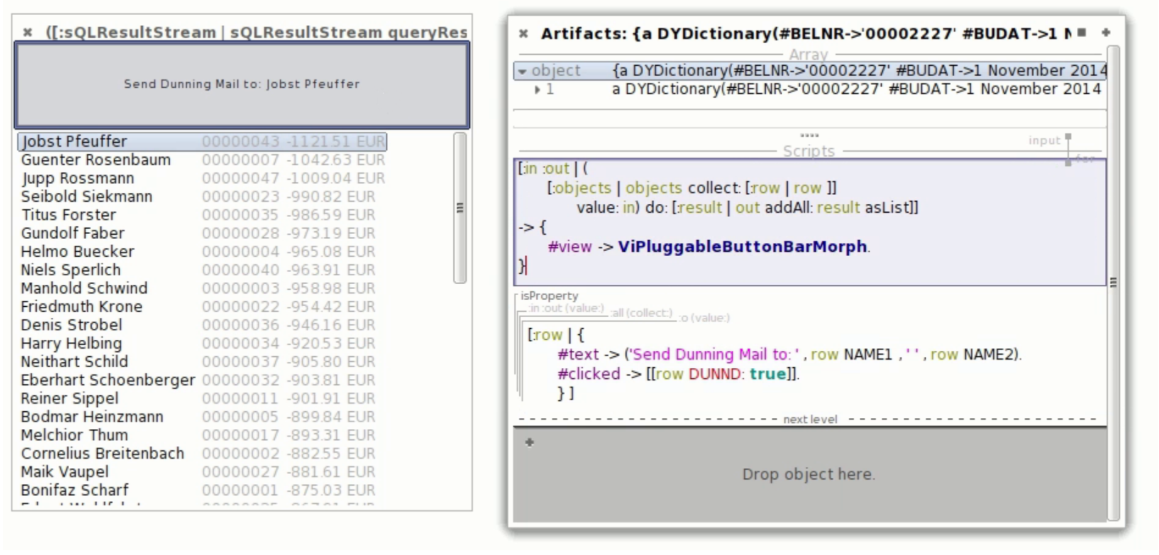
Figure 4: Screenshots of four versions of the dunning tool each showing the tool on the left and the corresponding script editor on the right. All four steps use the same SQL query result object.



```
[]: a SQLResultStream (SQLResultStream)
Jobst Pfeuffer          00000043 -1121.51 EUR
Guenter Rosenbaum       00000007 -1042.63 EUR
Jupp Rossmann           00000047 -1009.04 EUR
Seibold Siekmann        00000023 -990.82 EUR
Titus Forster           00000035 -986.59 EUR
Gundolf Faber           00000028 -973.19 EUR
Helmo Buecker           00000004 -965.08 EUR
Niels Sperlich          00000040 -963.91 EUR
Manhold Schwind         00000003 -958.98 EUR
Friedmuth Krone         00000022 -954.42 EUR
Denis Strobel           00000036 -946.16 EUR
Harry Helbing           00000034 -920.53 EUR
Neithart Schild         00000037 -905.80 EUR
Eberhart Schoenberger   00000032 -903.81 EUR
Reiner Sippel           00000011 -901.91 EUR
Bodmar Heinzmann        00000005 -899.84 EUR
Melchior Thum           00000017 -893.31 EUR
Cornelius Breitenbach   00000002 -882.55 EUR
Maik Vaupel             00000027 -881.61 EUR
Bonifaz Scharf          00000001 -875.03 EUR
```

```
Artifacts: {a SQLResultStream}, []
Array
object    {a SQLResultStream}
  1       a SQLResultStream
Scripts

:in :out (value:) :objects (collect:)
[:sQLResultStream | sQLResultStream queryResults ]

:in :out (value:) :objects (collect:)
[:row | row DEFICIT: row CREDIT - row DEBIT]

:in :out (value:)
[:rows | rows sorted: [:a :b | a DEFICIT < b DEFICIT]]

isProperty
:in :out (value:) :all (collect:) :o (value:)
[:row | {
    #text -> (row NAME1 , ' ' , row NAME2)
    }]

isProperty
:in :out (value:) :all (collect:) :o (value:)
[:row | {
    #text -> row KUNNR
    }]

isProperty
:in :out (value:) :all (collect:) :o (value:)
```

(c) Sorting the list of customers by increasing balance



```
([:sQLResultStream | sQLResultStream queryRes
Send Dunning Mail to: Jobst Pfeuffer
Jobst Pfeuffer          00000043 -1121.51 EUR
Guenter Rosenbaum       00000007 -1042.63 EUR
Jupp Rossmann           00000047 -1009.04 EUR
Seibold Siekmann        00000023 -990.82 EUR
Titus Forster           00000035 -986.59 EUR
Gundolf Faber           00000028 -973.19 EUR
Helmo Buecker           00000004 -965.08 EUR
Niels Sperlich          00000040 -963.91 EUR
Manhold Schwind         00000003 -958.98 EUR
Friedmuth Krone         00000022 -954.42 EUR
Denis Strobel           00000036 -946.16 EUR
Harry Helbing           00000034 -920.53 EUR
Neithart Schild         00000037 -905.80 EUR
Eberhart Schoenberger   00000032 -903.81 EUR
Reiner Sippel           00000011 -901.91 EUR
Bodmar Heinzmann        00000005 -899.84 EUR
Melchior Thum           00000017 -893.31 EUR
Cornelius Breitenbach   00000002 -882.55 EUR
Maik Vaupel             00000027 -881.61 EUR
Bonifaz Scharf          00000001 -875.03 EUR
```

```
Artifacts: {a DYDictionary(#BELNR->'00002227' #BUDAT->1 N
Array
object    {a DYDictionary(#BELNR->'00002227' #BUDAT->1 November 2014
  1       a DYDictionary(#BELNR->'00002227' #BUDAT->1 November 2014
Scripts
[:in :out | (
    [:objects | objects collect: [:row | row ]]
        value: in) do: [:result | out addAll: result asList]]
-> {
    #view -> ViPluggableButtonBarMorph.
}

isProperty
:in :out (value:) :all (collect:) :o (value:)
[:row | {
    #text -> ('Send Dunning Mail to: ' , row NAME1 , ' ' , row NAME2).
    #clicked -> [[row DUNND: true]].
    } ]
- - - - - - - - - - - - next level - - - - - - - - - - - -
Drop object here.
```

(d) Adding a button for sending out an email

Figure 4: Screenshots of four versions of the dunning tool each showing the tool on the left and the corresponding script editor on the right. All four steps use the same SQL query result object.

which allows users to manually determine which customers to send notices to. The example data we use is based on actual SAP ERP database tables. All columns contain the values encoded as strings. The example data is generated but adheres to the actual domains of the columns.

The dunning process is a data-intensive business process, as it involves collecting all invoices and payments from each customer and summing them up against each other. Only then can the overdue customers be selected [18].[1]

4.1.1 Creating the List of Debitors. Given that we already have an SQL query to determine the list of debitors, we can begin developing the tool by executing the dunning query in the SQL workspace (the SQL workspace in Figure 2 shows the query). As a result, we get a SQL result object which contains a sample of the query results which is already updating in the background. We now open a new Vivide pane and connect the outgoing port of the SQL workspace to the ingoing port of the empty pane and open the script editor. In the new script for the pane, we add one script to get the sample results from the query result object:

```
[:row | sqlResultStream queryResults]
```

We then add another script which creates a text property for each row in the result which states that the text of a list item should be the full name of the customer (see Figure 4a):

```
[:row | { #text
  -> (row NAME1 , ' ' , row NAME2) }]
```

Based on this initial view of the dunning tool we can now iterate the tool until it includes all relevant information: customer name, customer id, and balance (see Figure 4b). To make it easier to spot the customer with the most debit, we also sort the list increasingly according to their deficit, which brings the customers with the highest debt on top (see Figure 4c):

```
[:row | row DEFICIT: row CREDIT - row DEBIT]
[:rows | rows sorted:
  [:a :b | a DEFICIT < b DEFICIT]]
```

4.1.2 Sending out Dunning Notices. The list should also contain a way to interactively send out dunning notices. Therefore, we want to add a button for sending out a dunning email. As a first step, we create a new script, which we then add to the existing dunning list pane. Thereby, the list pane becomes a multi pane, which is a pane of panes. In there we connect the outgoing port of the dunning list with the ingoing port of the new empty script. Thereby, the new script receives the selection in the dunning list as its input and we have an example with which we can develop the button to send of dunning notices.

In order to display buttons, we have to change the view of the pane from a standard list view, to a `ViPluggableButtonBarMorph`. We can then define the appearance and the functionality of the button (see Figure 4d). For this demonstration we have left out the details of sending dunning notices:

```
[:row | {
  #text -> ('Send Dunning Mail to: ' ,
    row NAME1 , ' ' , row NAME2).
  #clicked -> [[
    "... Send out notice ..."
    row DUNND: true]]
}]
```

At this point, we can select a customer from the list and send a dunning notice to the customer by clicking the button. To update the list of customers to be inspected we have to notify the list of customers that there has been a change. Therefore, we set the list script to listen for the `#dunningListChanged` event which we send out on clicking the notice button:

```
...
#clicked -> [[
 "... Send out notice ..."
 row DUNND: true.
 ViEventNotifier
   trigger: #dunningListChanged]]
...
```

4.1.3 Wrapping up the Tool. We have developed the tool based on the query result object originating from the SQL workspace. To make the tool self-contained, we have to make it independent from this initial input. The query result object provides a method to re-execute the underlying query and be filled anew with the new query results. To expose this functionality to the user, we create another button. It receives the complete collection of database records from the customer list, picks any, determines the source result object, and calls `refresh` on it. At this point, the dunning tool is self-contained and does not require the original SQL workspace anymore.

4.2 Building a Histogram

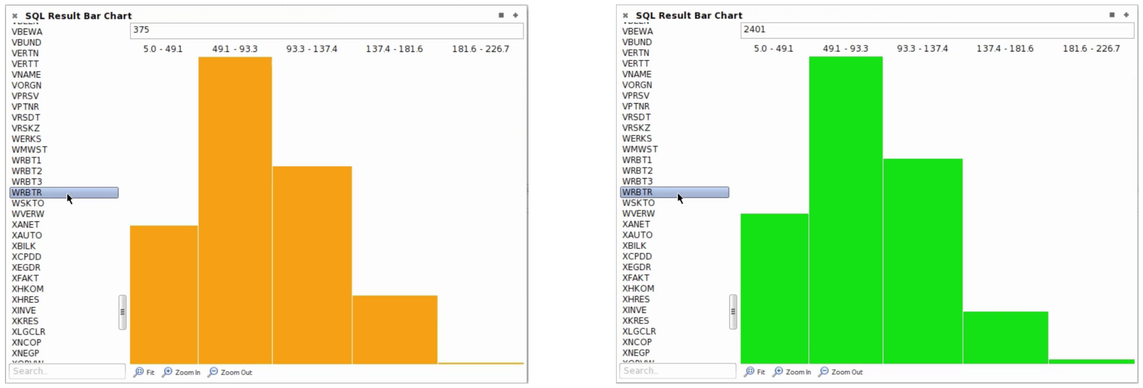
Histograms can be used to get an overview of the value distribution of large query results. The histogram tool illustrates how the same tool building techniques used for creating a business application can also be used for creating a more generic tool. The tool is based upon the capability of the query result object to return the count of distinct values per column.

The histogram (see Figure 5) consists of three parts: a list of field names, a bar chart, and a display of the number of rows used for the bar chart. The tool accepts a query result object as its input. The bars of the chart are colored in orange as long as the histogram is based on a sample of the data. In the background the aggregation keeps running and the bar chart regularly updates to provide the user with incremental feedback. When the histogram is showing the complete value distribution, the bars turn green (see Figure 5b).

The histogram tool first passes the incoming query result object to the list of field names which takes the first record from the result object and displays its field names:

```
[in first getSamples
  ifEmpty: [#()]
  ifNotEmptyDo: [:samples | samples first]]
value in: [:result |
 out addAll: result asList

[out addAll:
 ([:records | (records
```

[1]See the screencast at https://vimeo.com/202395765 to get an impression of the the interactive nature of the tool building process (In case any source code in the screencast differs from the described code, the described version is the current one).



(a) **(b)**

Figure 5: Screenshots of the histogram tool developed in the environment. It streams results and incrementally improves the counts of single buckets. When the histogram displays the complete distribution it turns green. In this case the initial histogram (a) is already very close to the complete version (b).

```
  ifEmpty: [#()]
  ifNotEmptyDo: [:theRecords |
   theRecords first keys
    collect: [:k | k asSymbol]])
 sorted]

[:symbol |
 { #text -> symbol asString }]
```

The pane on the right displays its data as a bar chart and accepts two inputs: the query result and the selected field name from the list of field names. The script of the bar chart, asks the query result for the distribution of values for the selected field name. The resulting `DYHistogram` object is then asked for a number of buckets, in our case 5.

```
[:results :fieldName |
 | buckets|
 buckets :=
  (results getDistributionFor: fieldName)
    bucketsForResolution: 5.
 { buckets .
   results getResultCount .
   (buckets collect: [:b | b count]) sum
 } asTuples ]
```

The resulting tuples consist of one bucket, the overall result count, and the sum of all elements currently in the buckets. These tuples are then mapped to view configurations for the single bars in the bar chart.

```
[:bucket :resultSum :bucketSum | {
  #value -> bucket count .
  #balloonText -> bucket count asString.
  #color -> (resultSum = bucketSum
    ifTrue: [Color green paler]
    ifFalse: [Color orange darker paler]) .
  #text -> bucket bucketLabel}]
```

The panel displaying the current number of records used for calculating the histogram gets the result object as its input. It directly displays the sum of the count of records of all buckets.

So far, the histogram would only display the bar chart once on selecting a field in the list of field names. To make use of the streaming of results, the script also has to sign up for the events generated by the query result object similar to the way done in the dunning tool.

4.3 Discussion

The proposed tools and mechanisms target the exploratory phase of business application development. While the developed applications can also be used in production settings, the resulting deliverable would have to be adapted to fit into the overall development processes of business applications. For example, the application might have to undergo a security audit or be adjusted to fit the user interface style guide of the overall system. Further, the environment currently supports cheap tool adaptation which might not be desirable in a business application scenario. Consequently, it might be required to lock the final version of a tool for deployment. This could easily be added to our approach.

The short feedback cycle in the environment is made possible by the two mechanisms of sampling and streaming. While they support a shorter feedback loop between developers and customers, they can also deceive developers regarding the correctness of the program or its responsiveness. We have implemented the sampling mechanism to return records as fast as possible. As a result, the sample is not representative, as this would require waiting for the complete query to finish. Such a sample could lead developers to make biased assumptions about the nature of the data, for example regarding corner cases, or the general domain of a field of a result. Further, they might also be mislead regarding the overall size of the result set. This trade-off could be mitigated by using database-provided sampling procedures which are provided by some databases. Similarly, the streaming mechanism is designed to mask the complete query execution duration. Thus, developers might underestimate the query execution time which will occur when the tool is waiting for the complete query result. This might however, not be relevant to all applications but only those which require the whole query to complete before the displayed data becomes useful.

5 RELATED WORK

The mechanism of streaming is implemented in a variety of languages and environments. For example, Orc [12] is a language whose abstractions are primarily build around streams. Similarly, environments such as ActiveSheets [28] enable non-professional programmers to work with streams of data in an interactive fashion. In contrast, the streaming mechanism used in our environment is only used to update a materialized collection of the stream results. The clients of this collection are notified trough an event system.

Tools such as Tempe [4] or Sintr [3] use similar mechanisms but target data analysis as the central use case. Tempe allows users to analyze different data sets in an interactive manner. It is based on LINQ [14] to query tables or streams of data and, as a result, can provide continuously updated visualizations of incoming data in streams. Sintr is designed for creating map-reduce programs for massive data sets. It uses streaming to provide developers with incremental feedback on the results. The mechanisms used in our approach are inspired by these systems and adapted to the context of application development.

End-user programming environments such as Excel [15] or DabbleDB [22] allow for the construction of applications based on relational data. Both support the processing of large data-sets to some extent but are also not designed as general application development environments. Gneiss [2] takes the spreadsheet approach further by also enabling streaming of external data and by continuously visualizing the information stored in the spreadsheets.

Other approaches start of from the static representation of the program behavior and enrich the development experience with exemplary data in an approach called "Programming with Examples" [11]. The approach tries to improve on the programming experience by integrating application-specific examples and tools into the development environment.

The ABAP workbench [1] development environment is designed for developing business application on top of relational databases. The development environment is a view on the code objects stored in the database of the ERP system. As a result of this direct integration of the development tools and the database, developers can access, inspect, and manipulate database data easily. However, regarding modifications of the application, ABAP is not specifically designed for short feedback cycles.

6 CONCLUSION

We presented a new approach to develop data-intensive business applications in a live programming environment supporting streamed access of sampled data. The illustration of the development of a dunning process tool and a histogram tool hint that the development is interactive and productive. With the help of a scripting language and graphical tools for editing the application, programmers can easily adapt the application while clarifying the requirements. Through this, mistakes in the application design can be discovered early in the process. If the application domain works with large amounts of data, long feedback loops do not have to be a necessary consequence. Even though live programming environments might have difficulties to process all the data available, our strategy of sampling and streaming could represent a productive trade-off.

Future Work. As the current implementation of the environment is only prototypical numerous challenges remain.

The current implementation of the mechanisms is limited as it only supports one initial query. Ideally, any subsequent collection protocol calls (for example filtering or mapping) should not be executed on the query result in the database. If the collection protocol execution requires features of the host language, then it should at least run in the remote process. Therefore, the Smalltalk collection protocol would require a back end similar to the LINQ implementation [14]. Further, some queries only produce a result on completion, for example the computation of an average. To provide fast feedback on such queries, any streaming mechanism must be able to return intermediate results.

Further, the mechanisms for exploring production data could be improved. While distributions already help with finding outliers, better tools are needed to identify outliers which only occur seldom. For example, the first name field of a table storing information on persons might contain variations of the string `unknown` to represent an unknown first name. These might not be obvious from the sample or a histogram.

Besides these technical challenges, it might be worthwhile researching ways to make the proposed means accessible to non-professional programmers. Processes which have previously been manually implemented by copying data from the database to spreadsheets might be implemented by users themselves. One first step would be to make the language for creating view configurations more accessible by making it more task-specific and closer to the vocabulary of business application users [16].

ACKNOWLEDGMENTS

We appreciate the knowledge exchange with the SAP Innovation Center Potsdam.[2] Sincere thanks also go to all PX workshop participants, who provided valuable feedback by discussing this topic thoroughly. We gratefully acknowledge the financial support of HPI's Research School[3] and the Hasso Plattner Design Thinking Research Program.[4]

REFERENCES

[1] Reudiger Buck-Emden and Jurgen Galimow. 1996. *SAP R 3 System: A Client/Server Technology (first edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. In *Proceedings of the ACM Symposium on User Interface Software and Technology, UIST 2014.* 87–96. DOI: http://dx.doi.org/10.1145/2642918.2647371

[3] Luke Church, Mariana Mărășoiu, and Alan Blackwell. 2016. Sintr: Experimenting with liveness at scale. (July 2016). https://www.cl.cam.ac.uk/~mcm79/pdf/2016-LIVE-Church-Marasoiu-Blackwell.pdf Proceedings of the second LIVE workshop on live programming systems (http://2016.ecoop.org/track/LIVE-2016).

[4] Danyel Fisher, Badrish Chandramouli, Rob DeLine, Jonathan Goldstein, Andrei Aron, Mike Barnett, John Platt, James Terwilliger, and John Wernsing. 2014. *Tempe: An Interactive Data Science Environment for Exploration of Temporal and Streaming Data.* Technical Report. https://www.microsoft.com/en-us/research/publication/tempe-an-interactive-data-science-environment-for-exploration-of-temporal-\and-streaming-data/

[2]icn.sap.com

[3]www.hpi.uni-potsdam.de/research_school

[4]www.hpi.de/en/research/design-thinking-research-program

[5] Heinz Forsthuber and Jrg Siebert. 2009. *SAP ERP Financials User's Guide*. SAP PRESS.

[6] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

[7] Christopher M. Hancock. 2003. *Real-Time Programming and the Big Ideas of Computational Literacy*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[8] Kyung-Kwon Hong and Young-Gul Kim. 2002. The critical success factors for ERP implementation: an organizational fit perspective. *Information & Management* 40, 1 (2002), 25–40. DOI: http://dx.doi.org/10.1016/S0378-7206(01)00134-3

[9] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan C. Kay. 1997. Back to the Future: The Story of Squeak - A Usable Smalltalk Written in Itself. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA) 1997*. 318–326. DOI: http://dx.doi.org/10.1145/263698.263754

[10] Dan Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. 2008. The Lively Kernel: A Self-supporting System on a Web Page. In *Proceedings of the Workshop on Self-Sustaining Systems (S3) 2008*. Springer, 31–50.

[11] Jun Kato, Takeo Igarashi, and Masataka Goto. 2016. Programming with Examples to Develop Data-Intensive User Interfaces. *IEEE Computer* 49, 7 (2016), 34–42. DOI: http://dx.doi.org/10.1109/MC.2016.217

[12] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. 2009. The Orc Programming Language. In *Proceedings of FMOODS/FORTE 2009* (Lisbon, Portugal, 9–11 Jun 2009) *(Lecture Notes in Computer Science)*, David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter (Eds.), Vol. 5522. Springer, 1–25. DOI: http://dx.doi.org/10.1007/978-3-642-02138-1_1

[13] Thor Magnusson. 2014. Herding Cats: Observing Live Coding in the Wild. *Computer Music Journal* 38, 1 (2014), 8–16. DOI: http://dx.doi.org/10.1162/COMJ_a_00216

[14] Erik Meijer, Brian Beckman, and Gavin M. Bierman. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the Conference on Management of Data (SIGMOD) 2006*. 706. DOI: http://dx.doi.org/10.1145/1142473.1142552

[15] Microsoft Corporation. 2017. Excel. (3 Feb. 2017). https://products.office.com/excel

[16] Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA.

[17] Donald A. Norman and Stephen W. Draper. 1986. *User Centered System Design*. Lawrence Erlbaum Associates, Inc., Publishers.

[18] Hasso Plattner and Bernd Leukert. 2016. *The In-Memory Revolution: How SAP HANA Enables Business of the Future*. Springer International Publishing.

[19] Patrick Rein, Stefan Lehmann, Toni, and Robert Hirschfeld. 2016. How Live Are Live Programming Systems?: Benchmarking the Response Times of Live Programming Environments. In *Proceedings of the Programming Experience Workshop (PX/16) 2016 (PX/16)*. ACM, New York, NY, USA, 1–8. DOI: http://dx.doi.org/10.1145/2984380.2984381

[20] August-Wilhelm Scheer and Frank Habermann. 2000. Enterprise Resource Planning: Making ERP a Success. *Commun. ACM* 43, 4 (April 2000), 57–61. DOI: http://dx.doi.org/10.1145/332051.332073

[21] Ben Shneiderman, Catherine Plaisant, Maxine Cohen, and Steven Jacobs. 2009. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (international edition of 5th revised edition ed.). Pearson, Upper Saddle River, New Jersey, USA.

[22] Smallthought Systems Inc. 2017. DabbleDB. (3 Feb. 2017). http://blog.dabbledb.com/

[23] Deborah Szebeko and Lauren Tan. 2010. Co-designing for Society. *AMJ* 3, 9 (2010), 580–590.

[24] Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. 2014. Interleaving of Modification and Use in Data-Driven Tool Development. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!) 2014*. ACM, 185–200.

[25] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013*. San Francisco, California, USA, 31–34. DOI: http://dx.doi.org/10.1109/LIVE.2013.6617346

[26] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (1997), 38–43. DOI: http://dx.doi.org/10.1145/248448.248457

[27] David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Proceedings of Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) 1987*. ACM, New York, New York, USA, 227–242.

[28] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. 2014. Stream Processing with a Spreadsheet. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2014*. Springer, 360–384. DOI: http://dx.doi.org/10.1007/978-3-662-44202-9_15