Aggregating and Disaggregating Flexibility Objects

Laurynas Šikšnys, Emmanouil Valsomatzis, *Student Member, IEEE*, Katja Hose, and Torben Bach Pedersen, *Senior Member, IEEE*

Abstract—In many scientific and commercial domains, we encounter flexibility objects, i.e., objects with explicit flexibilities in a time and an amount dimension (e.g., energy or product amount). Applications of flexibility objects require novel and efficient techniques capable of handling large amounts of such objects while preserving flexibility. Hence, this paper formally defines the concept of flexibility objects (flex-objects) and provides a novel and efficient solution for aggregating and disaggregating flex-objects. Out of the broad range of possible applications, this paper will focus on smart grid energy data management and discuss strategies for aggregation and disaggregation of flex-objects while retaining flexibility. This paper further extends these approaches beyond flex-objects originating from energy consumption by additionally considering flex-objects originating from energy production and aiming at energy balancing during aggregation. In more detail, this paper considers the complete life cycle of flex-objects: aggregation, disaggregation, associated requirements, efficient incremental computation, and balance aggregation techniques. Extensive experiments based on real-world data from the energy domain show that the proposed solutions provide good performance while satisfying the strict requirements.

Index Terms—Aggregation, incremental aggregation, balance aggregation, disaggregation, flex-objects

1 Introduction

Many scientific and commercial domains deal with flexibilities in terms of *time* and *amount*. In the *smart-grid* domain, for instance, the EU FP7 research project MIRABEL [1] and the ongoing Danish project TotalFlex (www.totalflex.dk) exploit *time* and *energy amount* flexibilities of various electricity consumers and producers to increase the share of renewable energy sources (RES) such as wind-turbines and solar-panels.

Such flexibilities can be captured by *flexibility objects* (in short, *flex-objects*), specifying (1) *how much energy* is needed, (2) *when* it is needed, and (3) what the *tolerated flexibilities* are regarding *time* (e.g., between 9 PM and 5 AM) and *energy amount* at consecutive time intervals (e.g., between 2 and 4 kWh in the first hour and 3 and 5 kWh in the second hour). By interconnecting thousands of european electricity market participants—*consumers*, *producers*, *aggregators*, and *balance responsible parties* (BRPs)—using a large-scale energy data management system [1], these flex-objects undergo the cycle of *aggregation*, *instantiation*, and *disaggregation*, depicted in Fig. 1. Generated by individual consumers/producers (e.g., a smart-home automation system for charging the battery of an electric vehicle (EV)), flex-objects ($f_1, \ldots, f_4$) are sent to aggregators, which first group similar (time-overlapping) flex-objects (g_1 and g_2) and then aggregate them into larger "macro" flex-objects (f_{a1} and f_{a2}). Utilizing such "macro" flex-objects, the BRP schedules (globally balances) flexible loads of the "macro" flex-objects to match the forecasts of inflexible consumption and RES production. During the scheduling, the "macro" flex-objects are transformed (*instantiated*) into so-called "macro" *fix-objects* (f^x_{a1} and f^x_{a2}) having concrete values assigned for the time and amount dimensions within the flex-object flexibility intervals. Then, the aggregators disaggregate the "macro" fix-objects into "micro" fix-objects ($f^x_1, \ldots, f^x_4$), which specify the exact time and magnitude of energy that has to be consumed (or produced) by the consumers/producers. If electricity is consumed and produced according to the fix-objects, consumers/producers are ultimately rewarded based on the flexibility they offer.

In this cycle, the aggregation and disaggregation operations have a number of associated requirements. First, aggregation must reduce the total number of flex-objects to lower the complexity of solving the scheduling problem. It should retain as much flexibility as possible in the aggregated flex-objects, while not introducing more flexibility than that of the non-aggregated flex-objects. Second, the aggregation must produce aggregated flex-objects conforming to individual BRP requirements, setting the limits for, for instance, the number, magnitude of amount, and covered time window, of the flex-objects. Third, aggregation must be able to support rapid and continuous flex-object additions and removals issued by consumers/producers. Fourth, to reduce the risk of congestions (caused by RES and EVs), it is very important to balance demand and supply locally by aggregators prior to scheduling (global balancing) by BRPs. Thus, the aggregation must be able to perform local balancing when producing aggregated flex-objects. Fig. 1 demonstrates such "balance aggregation" using the flex-object f_{a1}, which represents the joint energy needs/offers from a consumer issuing the flex-object f_1 and a producer issuing the flex-object f_2. Lastly, the disaggregation of fix-objects must be feasible in all these

• *The authors are with the Department of Computer Science, Aalborg UniversityDK-9220, Aalborg Øst, Denmark.*
E-mail: {siksnys, evalsoma, khose, tbp}@cs.aau.dk.

Manuscript received 4 June 2014; revised 31 Mar. 2015; accepted 28 May 2015. Date of publication 14 June 2015; date of current version 2 Oct. 2015.
Recommended for acceptance by C.-Y. Chan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2015.2445755

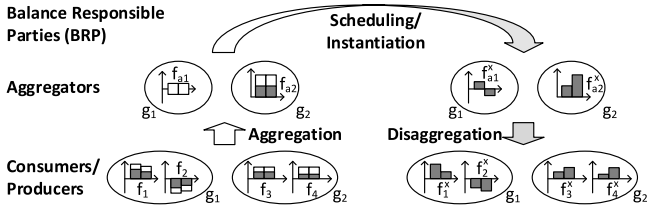


Fig. 1. The lifecycle of flex-objects.

cases. For use in the smart-grid (MIRABEL/TotalFlex) and other domains, no flex-object aggregation/disaggregation solution satisfying all these requirements exists.

Considering this need, the contributions of this paper are as follows. First, we formally define flex-objects, fix-objects, measures to quantify flexibility, aggregation and disaggregation functions, and associated requirements. Second, we present a basic technique to aggregate many flex-objects into a single aggregated flex-object and to disaggregate a fix-object into many fix-objects. Third, we present an advanced technique that generates many aggregated flex-objects and that is able to disaggregate fix-objects. Here, the aggregation is performed incrementally. Given a sequence of flex-object deltas, our technique partitions flex-objects into disjoint groups of similar flex-objects. The partitioning is performed in two steps—grid-based *grouping* and *bin-packing*—ensuring that flex-objects in the groups are similar enough and that the groups themselves fulfil a given (aggregate) criterion. After bin-packing, similar flex-objects from different groups are merged into aggregated flex-objects, which are finally returned as output in the form of a sequence of aggregated flex-object deltas. Fourth, we present five techniques for balance aggregation. For all techniques, we provide detailed algorithms in pseudo-code along with computational complexity estimates. Finally, we discuss the results of our extensive set of experiments with both regular and balance aggregation and show that our solution scales well and handles aggregation and disaggregation efficiently and effectively.

This paper significantly extends our previous work [2] by providing (1) a more concise problem formulation, also including balance aggregation, (2) detailed aggregation/disaggregation algorithms with pseudo-code, (3) five concrete balance aggregation algorithms, (4) computational complexity analyses, and (5) a set of comprehensive balance aggregation experiments.

The remainder of the paper is structured as follows. Section 2 formally defines all relevant concepts. Sections 3–4 describe basic and advanced flex-object aggregation/disaggregation techniques. Balance aggregation techniques are introduced in Section 5. Section 6 describes the experimental evaluation, while Section 7 discusses related work. Finally, Section 8 concludes the paper and discusses future work. An electronic appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2014.2445755, provides additional proofs and the detailed complexity analyses of the proposed algorithms.

2 Problem Formulation

We now formalize the problem of aggregating and disaggregating flexibility objects. Our formalization includes

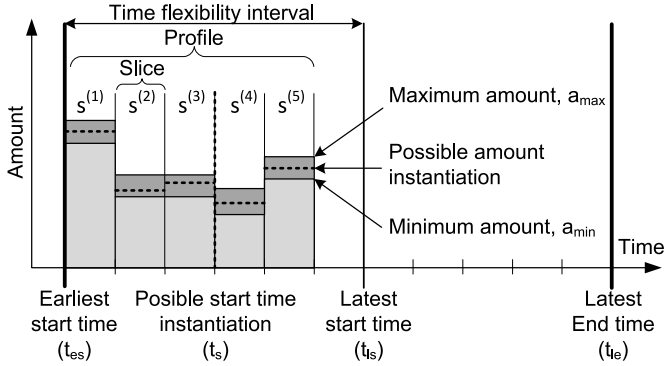


Fig. 2. A generic flex-object.

(1) definitions of flex-object concepts, (2) measures for quantifying flexibility, and (3) functions for *aggregation* and *disaggregation* and their associated constraints.

Let e be an *entity* from a *domain* D utilizing a *resource* r; the utilization of the resource is characterized by a *continuous amount* over a *discrete time*. An example of an entity (e) is an EV that is connected to a charging station (D) and consuming specific amounts of electrical energy (r) at different hours to fully charge its battery. We define flexibility in how the resource can be utilized (i.e., flexible utilization) using a so-called *flex-object*, which is a multidimensional object capturing two aspects: (1) a *time flexibility interval* and (2) an *amount profile* with a sequence of consecutive *slices*, each defined by minimum and maximum bounds of the amount.

Definition 1. *A* flex-object f *is a tuple* $f = ([t_{es}, t_{ls}], p)$ *where* $[t_{es}, t_{ls}]$ *is the* start time flexibility interval *and* p *is the* amount profile. *The time is discretized into equal-sized units, e.g., 15 minute intervals. Thus, we use* $t_{es} \in \mathbb{N}$ *to specify the* earliest start time *and* $t_{ls} \in \mathbb{N}$ *to specify the* latest start time. *The* p *is a sequence of slices* $\langle s^{(1)}, \ldots, s^{(m)} \rangle$, *where a* slice $s^{(i)}$ *is a continuous range* $[a_{min}, a_{max}]$ *defined by a* minimum amount a_{min} *and a* maximum amount a_{max}. *The extent of* $s^{(i)}$ *in the time dimension is 1 unit. Hence, a flex-object's* profile duration *is computed as* $p_{dur}(f) = |f.p|$, *its* earliest end time *as* $t_{ee}(f) = f.t_{es} + p_{dur}(f)$, *and its* latest end time *as* $t_{le}(f) = f.t_{ls} + p_{dur}(f)$.

Fig. 2 depicts an example of a flex-object specifying the intended consumption of electricity of a single EV connected to a charging station, where the EV, electricity, and charging station are the above presented entity e, resource r, and domain D, respectively. The flex-object has a profile with five slices: $\langle s^{(1)}, \ldots, s^{(5)} \rangle$. Every slice is represented by a bar in the figure. The top of the light-shaded bar represents the minimum amount value (a_{min}) and the top of the dark-shaded bar represents the maximum amount value (a_{max}).

Depending on the values of the amount bounds, we distinguish the following three types of flex-objects:

Definition 2. *A flex-object* f *is called* positive *if* $\forall s \in f.p : s.a_{min} \geq 0$. *A flex-object* f *is called* negative *if* $\forall s \in f.p : s.a_{max} < 0$. *A flex-object* f *is called* mixed *if it is neither* positive *nor* negative, *i.e., if* $\exists s_n, s_p \in f.p : s_n.a_{min} < 0 \wedge s_p.a_{max} \geq 0$.

For example, a *positive* flex-object can be used to describe the consumption profile and associated flexibilities of a

heat-pump heating a house. A *negative* flex-object can be used for various production-only power systems, e.g., solar panels, wind-turbines, or power generators. A mixed electrical behaviour is exhibited by more advanced power systems, e.g., EVs whose batteries can be charged or discharged. A *mixed* flex-object can be used to capture such specific behaviour.

We distinguish two types of flexibilities associated with f. The *time flexibility* $tf(f)$ is the difference between the latest and earliest start time, i.e., $tf(f) = f.t_{ls} - f.t_{es}$. Similarly, the *amount flexibility* $af(f)$ is the sum of the differences between the amount bounds of all slices in f's profile, i.e., $af(f) = \sum_{s \in f.p} s.a_{max} - s.a_{min}$.

A flex-object with time and amount flexibilities equal to zero is called a *fix-object*. In this case, the fix-object $f = ([t_{es}, t_{ls}], p)$ is such that $t_{es} = t_{ls}$ and $\forall s \in f.p : s.a_{min} = s.a_{max}$. A fix-object may or may not be a *valid instance* of a given flex-object.

Definition 3. *A valid* instance *(instantiation) of a flex-object* $f = ([t_{es}, t_{ls}], \langle s^{(1)}, \ldots, s^{(m)} \rangle)$ *is a fix-object* $f^x = ([t_s, t_s], \langle s_x^{(1)}, \ldots, s_x^{(m)} \rangle)$ *such that* $t_{es} \leq t_s \leq t_{ls}$ *and* $\forall i = 1..m : s^{(i)}.a_{min} \leq s_x^{(i)}.a_{min} = s_x^{(i)}.a_{max} \leq s^{(i)}.a_{max}$. *We use the notation* $f^x \rhd f$ *to denote that a fix-object* f^x *is a valid instance of a flex-object* f. *We refer to* t_s *as the (assigned)* start time.

In the general case, there is an infinite number of possible instances of a flex-object ($f^x \rhd f$). One possible instance is shown as the dotted line in Fig. 2. To quantify the size of the possible instantiation space (flexibility), we define *total flexibility* as follows:

Definition 4. *The total flexibility of a flex-object* f *is the product of* time *and* amount flexibility, *i.e.,* $flex(f) = tf(f) \cdot af(f)$.

A flex-object with a larger *total flexibility* represents a larger variety of instantiations compared to a flex-object with lower *total flexibility*. Consider a flex-object $f = ([2, 7], \langle s^{(1)}, s^{(2)} \rangle)$ where $s^{(1)} = [10, 20]$ and $s^{(2)} = [18, 30]$. The time flexibility of f is equal to $7 - 2 = 5$. The amount flexibility $af(f)$ is equal to $(20 - 10) + (30 - 18) = 22$. Hence, the total flexibility of f is equal to 110, and it is considered "more flexible" than, for example, a flex-object with a total flexibility of 100.

We now generally define the concepts of flex-object *aggregation* and *disaggregation*, and formulate associated constraints and requirements. Later, in Sections 3–5, we elaborate on how these operations are performed.

Definition 5. *Flex-object* aggregation *generalizes the joint flexible utilization of a resource within the domain. This is performed by a function* $AGG(F)$ *that takes a set of flex-objects* F *and produces a set of flex-objects* A, $|A| \leq |F|$. *Every* $f_a \in A$ *is called an* aggregated flex-object.

Definition 6. *Flex-object* disaggregation *de-generalizes (coarse) instances of aggregated flex-objects by generating (detailed) instances of non-aggregated flex-objects. This is performed by a function* $DAGG(F, A, A^X)$ *where* A^X *is a set of fix-objects that are instances of the aggregated flex-objects from* A *such that* $A = AGG(F) \wedge \forall f_a \in A : \exists f_a^x \in A^X$ *and* $f_a^x \rhd f_a$. *The function produces a set of non-aggregated flex-object instances,* F^X, *such that* $\forall f \in F : \exists f^x \in F^X$ *where* $f^x \rhd f$.

There exist many different pairs of aggregation and disaggregation functions (AGG, $DAGG$). However, we are primarily interested in those pairs that "correctly summarize" amounts (and their allocations in time). We formulate this as the following requirement.

Amount conservation requirement. For any given F, A^X, and $F^X = DAGG(F, AGG(F), A^X)$, the following equality must hold for every time instance $t \in \mathbb{N}$:

$$\sum_{f_a^x \in A^X} \sum_{i=1}^{|f_a^x.p|} \left[f_a^x.p[i].a_{min} | t = f_a^x.t_{es} + i \right] = \quad (1)$$

$$\sum_{f^x \in F^X} \sum_{i=1}^{|f^x.p|} [f^x.p[i].a_{min} | t = f^x.t_{es} + i]. \quad (2)$$

The conservation requirement ensures that the amounts of flex-object instances are equal before and after (dis-) aggregation at all time intervals. In addition, individual aggregation and disaggregation functions must comply with a number of requirements that are inspired by the MIRABEL/TotalFlex use-cases, but are also important for flex-object aggregation in general:

Compression/flexibility trade-off requirement. AGG must allow controlling the trade-off between the number of aggregated flex-objects and the *flexibility loss*—the difference between the *total flexibility* (see Definition 4) before and after aggregation.

Aggregate constraint requirement. Every aggregated flex-object $f_a \in AGG(F)$ must satisfy a so-called *aggregate constraint* C, which is satisfied only if the value of a certain flex-object attribute, e.g., *total maximum amount*, is within the given bounds and thus the aggregated flex-objects are "properly shaped" to meet the BRP rules.

Incremental update requirement. Flex-object updates (addition/removal) should be processed efficiently and cause minimal changes to the set of aggregated flex-objects. This is vital in scenarios, like MIRABEL, where addition/removal of flex-objects are very frequent.

Balance requirement. When local balance is required, the aggregation function AGG must generate flex-objects by specifying the joint flexible resource utilization of counteracting entities within the domain D. Thus, AGG should minimize the sum of the *absolute balance* of the aggregated flex-objects, where the absolute balance of a flex-object f, $AbsBalance(f)$, is the sum of the absolute amount averages ($|\frac{a_{min}+a_{max}}{2}|$) of its slices.

This ensures that amount bounds of an aggregated flex-object are centred (balanced) around zero, as in $f_{a1} = ([1, 10], \langle [-10, +10], [-10, +10] \rangle)$ in Fig. 1. Such an aggregated flex-object is, potentially, *mixed* and captures (1) the balanced utilization of a resource within a domain D as a nominal option, and (2) feasible deviations from the balance, describing how the utilization can be unbalanced while respecting all flex-object constraints. In practice, such balance-aggregated flex-objects can be used by BRPs, for example, to balance demand and supply while charging a fleet of EVs and at the same time following price signals from the regulating power market.

Note that the *compression/flexibility trade-off* and *balance* requirements are conflicting, because it is not possible to

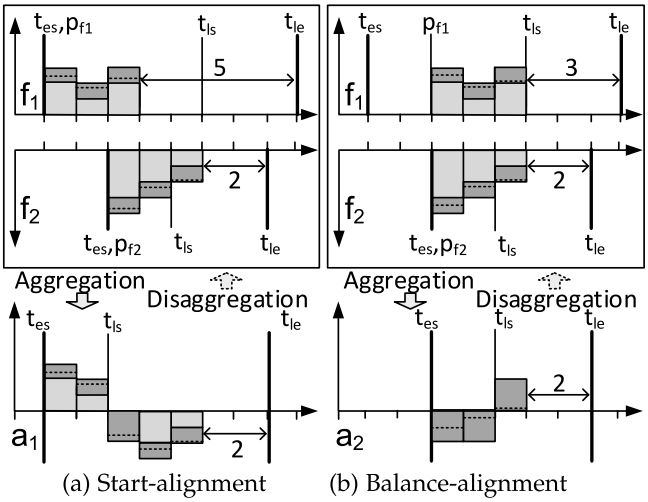


Fig. 3. N-to-1 aggregation using different profile alignment options, and the 1-to-N disaggregation.

achieve a small number of aggregated flex-objects with high flexibility and balance at the same time.

3 Aggregation and Disaggregation

In this section, we propose basic flex-object aggregation and disaggregation functions satisfying only the amount conservation requirement. As these two functions produce (consume) a single aggregated flex-object, we denote them as N-to-1 and 1-to-N, respectively. In Section 4, we generalize these functions for a larger set of aggregated flex-objects (N-to-M and M-to-N) and revisit the remaining requirements.

N-To-1 aggregation. As defined in Section 2, the flex-object profile is not fixed in time, but it must be positioned so that the earliest start time $f.t_{es}$ and the latest start time $f.t_{ls}$ bounds are respected. Hence, the aggregation of even two flex-objects is not straightforward. Consider aggregating two flex-objects f_1 and f_2 with time flexibilities 5 and 2. Thus, we have 18 $((5+1)\cdot(2+1))$ different profile *positioning* combinations, each of them realizing a different aggregated flex-object. Fig. 3 depicts two such combinations.

In the simplest case, flex-object profiles can be positioned at their earliest start time (t_{es}), see Fig. 3a. We refer to this type of positioning as *start-alignment (SA)*. In this case, Algorithm 1 is used to aggregate flex-objects from a set F into a single flex-object f_a (N-To-1).

Algorithm 1. N-To-1 Aggregation Using *Start-Alignment*

Input: F - a set of flex-objects;
Output: f_a - an aggregated flex-object;
1 **Function** *AGG-N-to-1*(F):
2 **for** $f \in F$ **do**
3 $p_f \leftarrow f.t_{es}$; //Start-aligning
4 **end**
5 $f_a.t_{es} \leftarrow min_{f\in F}(p_f)$;
6 $f_a.t_{ls} \leftarrow f_a.t_{es} + min_{f\in F}(f.t_{ls} - p_f)$;
7 **for** $t \in [min_{f\in F}(p_f)+1, max_{f\in F}(p_f + p_{dur}(f))]$ **do**
8 $s_a \leftarrow f_a.p[t - f_a.t_{es}]$; $s_a.a_{min} \leftarrow \sum_{f\in F} f.p[t-p_f].a_{min}$;
9 $s_a.a_{max} \leftarrow \sum_{f\in F} f.p[t-p_f].a_{max}$;
10 **end**
11 **return** f_a

First, Algorithm 1 partially instantiates flex-objects in F by choosing the absolute profile positions $p_{f_1}, \ldots, p_{f_{|F|}}$ using *start-alignment* (Lines 2-4). Then, the start time flexibility bounds of the aggregated flex-object are computed conservatively so that the aligned profiles of $f \in F$ can always be shifted within the flexibility range of f_a (Lines 5-6). Finally, the profile of the aggregated flex-object is constructed by summing up the minimum and maximum amounts of the slices of the aligned profiles, ignoring the slices with out-of-range indices (Lines 7-10).

In general, there are many other ways to align profiles by choosing the profile positions $p_{f_1}, \ldots, p_{f_{|F|}}$ at Lines 2-4. Each of these alignments determines where amounts from individual flex-objects are allocated within the profile of f_a. A few basic alignment options have already been presented and discussed [2]. Additionally, we propose a novel so-called *balance-alignment*, solving the non-linear balance optimization problem to find the profile positions, so that the amount bounds of aggregated flex-object are allocated around zero, as formulated by the *balance requirement*. The effects of the *start* and *balance* alignments are illustrated in Figs. 3a and 3b. The balance alignment is further elaborated on in Section 5.

1-To-N disaggregation. Given a set of flex-objects F, an aggregated flex-object $f_a = \$AGG\text{-}N\text{-}to\text{-}1(F)$, its instance f_a^x ($f_a^x \rhd f_a$), and the set of profile alignment positions $P_F = \{p_{f_1}, \ldots, p_{f_{|F|}}\}$, where P_F is $\{f_1.t_{es}, \ldots, f_{|F|}.t_{es}\}$ for *start-alignment*, the disaggregation of f_a^x is performed using Algorithm 2.

Algorithm 2. 1-To-N Disaggregation

Input: F - a set of flex-objects; f_a - an aggregated flex-object; f_a^x - an instance of f_a; $P_F = \{p_{f_1}, \ldots, p_{f_{|F|}}\}$ - a set of profile alignment positions
Output: F^X - a set of flex-object assignments
1 **Function** *DAGG-1-to-N*(F, f_a, f_a^x, P_F):
2 **for** $i \leftarrow 1..|f_a^x.p|$ **do**
3 $s^x \leftarrow f_a^x.p[i]$;
4 $s \leftarrow f_a.p[i]$;
5 $s^x.a_{min} \leftarrow s^x.a_{max} \leftarrow \frac{s^x.a_{min} - s.a_{min}}{s.a_{max} - s.a_{min}}$;
6 **end**
7 **for** $f \in F$ **do**
8 $f^x.t_{es} \leftarrow f^x.t_{ls} \leftarrow f_a^x.t_{es} - f_a.t_{es} + p_f$;
9 **for** $i \leftarrow 1..|f.p|$ **do**
10 $f^x.p[i].a_{min} \leftarrow f^x.p[i].a_{max} \leftarrow$
 $f.p[i].a_{min} + (f.p[i].a_{max} - f.p[i].a_{min}) \cdot$
 $(f_a^x.p[p_f - f_a.t_{es} + i].a_{min})$;
11 **end**
12 **end**
13 **return** $\{f_1^x, f_2^x, \ldots, f_{|F|}^x\}$

First, the algorithm normalizes the amounts of f_a^x, making them relative to the amount bounds of f_a (Lines 2-6). Then, it builds instances for every $f \in F$ so that their start-times are equally indented (Line 8) and the amounts are distributed proportionally, having the same relative allocations as the f_a^x amounts (Lines 9-12). The disaggregation of two aggregated flex-object instances (dashed lines) are shown in Figs. 3a and 3b.

The pair of aggregation and disaggregation functions (*AGG-N-to-1*, *DAGG-1-to-N*) satisfies the *amount conservation*

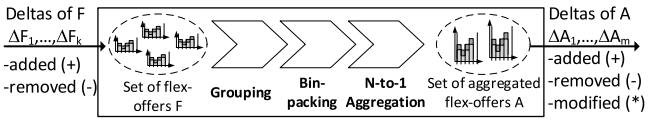


Fig. 4. N-to-M aggregation input, output, and phases.

requirement (the proof is given in Appendix A.1, available online). Therefore, for a given instance of an aggregated flex-object it is always possible to generate valid instances of non-aggregated flex-objects such that the amounts of fix-objects before and after disaggregation match. The N-to-1 aggregation with *start-alignment* and the 1-to-N disaggregation require $\mathcal{O}(|f_a.p| \cdot |F|)$ time, where $|f_a.p|$ is the length of the aggregated flex-object profile (the full analysis is given in Appendix A.2, available online).

To summarize, the N-to-1 aggregation and the 1-to-N disaggregation functions can be used to aggregate and disaggregate flex-objects while satisfying the *amount conservation* requirement. However, the aggregated flex-objects exhibit *time flexibility* losses, which depend on the alignment and the flex-object with the smallest time flexibility (see Lines 5-6 in Algorithm 1). To address this limitation and to meet the additional requirements from Section 2, we now propose N-to-M aggregation and M-to-N disaggregation approaches.

4 N-to-M Aggregation

As discussed in Section 3, aggregating flex-objects with different start time flexibilities may result in an unnecessary loss of time flexibility. This loss can be avoided by carefully grouping flex-objects and thus ensuring that their time flexibility intervals are similar (or equal). We now describe an incremental (N-to-M) approach that performs multiple levels of grouping to aggregate a set of flex-objects $\mathcal{F}$ into a set of aggregated flex-objects $\mathcal{A}$ while satisfying the *compression/flexibility trade-off*, *aggregate constraint*, and *incremental update* requirements.

4.1 Overview of the N-To-M Aggregation

As shown in Fig. 4, the N-to-M aggregation approach (internally) manages a set of non-aggregated flex-objects F and a set of aggregated flex-objects A. The sets F and A are maintained using sequences of deltas (updates). $\Delta F_1, \ldots, \Delta F_k$ is the input and $\Delta A_1, \ldots, \Delta A_m$ is the output of the aggregation. Each delta ΔF_i is of the form (f, c), where f is a non-aggregated flex-object and $c \in \{+, -\}$ indicates insertion ($+$) or deletion ($-$) of f in F. Similarly, a delta ΔA_i is of the form (f_a, c_a), where f_a is an aggregated flex-object and $c_a \in \{+, -, *\}$ indicates insertion ($+$), deletion ($-$), or modification (*) of f_a in A. To explain how the output $\Delta\ A_1, \ldots, \Delta A_m$ is generated from the input $\Delta F_1, \ldots, \Delta F_k$, we now provide a high-level and intuitive explanation of the *logical phases* of *grouping*, *bin-packing*, and *N-to-1 aggregation*.

4.2 Logical Phases of the N-To-M Aggregation

The input $\Delta F_1, \ldots, \Delta F_k$ is passed through the *logical* phases of (1) *grouping*, which partitions flex-objects (from F) into disjoint groups of similar flex-objects, (2) *bin-packing*, which packs flex-objects into smaller sub-groups to ensure an aggregation constraint (e.g., the total energy should be below $100\,kWh$), and (3) *N-to-1 aggregation*, which applies the N-to-1 aggregation function for each sub-group to

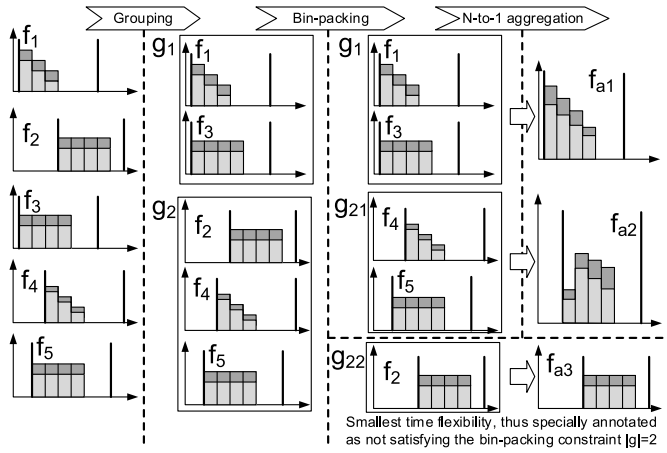


Fig. 5. Flex-object aggregation using the N-to-M approach.

produce aggregated flex-objects (see Fig. 5). Propagation of data is done incrementally, avoiding re-computation of groups, sub-groups, and aggregated flex-objects unaffected by the deltas $\Delta F_1, \ldots, \Delta F_k$.

Grouping phase. In this logical phase, the set of flex-objects F is maintained according to $\Delta F_1, \ldots, \Delta F_k$ and then incrementally partitioned into disjoint groups of similar flex-objects. For this purpose, flex-object *similarity grouping* is performed: two flex-objects are grouped together if the values of specific flex-object attributes, e.g., *earliest start time*, *latest start time*, and/or *time flexibility*, differ by no more than respective user-defined tolerance thresholds. The associated grouping attributes and user-defined tolerance thresholds are called *grouping parameters*. In the example in Fig. 5, flex-objects $f_1, f_2, \ldots, f_5$ are partitioned into the groups g_1 and g_2 during the grouping phase.

Bin-packing phase. This logical phase incrementally enforces the aggregate constraint (see Section 2). Each affected group g produced in the *grouping* phase is either passed to the next logical phase (if g satisfies the constraint already) or further partitioned into the minimum number of bins (groups) such that the constraint $w_{min} \leq w(b) \leq w_{max}$ is satisfied and time flexibility is retained by each bin b. Here, $w(b)$ is a weight function and w_{max} and w_{min} are the upper and lower bounds. We refer to w_{min}, w_{max}, and w as *bin-packing parameters*. By adjusting these parameters, groups with, for instance, a bounded number of flex-objects or a bounded total amount can be built. Note that it may be impossible to satisfy the constraint for certain groups. For example, consider a group with a single flex-object, while we impose a lower bound of two flex-objects in all groups. Aggregated flex-objects of the sub-groups failing to satisfy the aggregate constraint are specially annotated (see g_{22} in Fig. 5).

N-to-1 aggregation phase. For each of the affected bins, aggregated flex-objects are produced by applying an incremental variant of the N-to-1 aggregation (see Section 3). The alignment option (strategy) is specified as an *aggregation parameter*.

Fig. 5 visualizes the processing of five flex-object insert deltas $(f_1, +), \ldots, (f_5, +)$ in all three logical phases. In these phases, $f_1, \ldots, f_5$ are aggregated into the flex-objects f_{a1}, f_{a2}, and f_{a3}, which are ultimately packed into the insert deltas $(f_{a1}, +)$, $(f_{a2}, +)$, and $(f_{a3}, +)$ returned as output. In this example, grouping parameters are set so that the maximum

difference between the earliest start times (t_{es}) is at most 2. The bin-packing parameters are set so that resulting groups have exactly two flex-objects, i.e., $w_{min} = w_{max} = 2$, $w(g) = |g|$. In the N-to-1 aggregation phase, *start-alignment* is used.

Here, the formation of groups, sub-groups, and aggregates is configured using user-defined parameters—grouping, bin-packing, aggregation parameters. In practice, users are not required to set or fine-tune these (cumbersome) parameters. Instead, they will be offered a number of meaningful pre-defined parameter configurations corresponding to typical business requirements, e.g., *short (long) balanced profiles* or *amounts as early as possible and limited to 100*.

4.3 Algorithms for the N-To-M Aggregation

For realizing the N-to-M aggregation, we use the functions *initAgg*, *processDelta*, and *aggregateInc*. Algorithm 3 uses these functions to perform the N-to-M aggregation. The algorithm produces a set of aggregated flex-objects $F_A = \{a_1, \ldots, a_M\}$ given a set of non-aggregated flex-objects $F = \{f_1, \ldots, f_N\}$ and the discussed *grouping*, *bin-packing*, and *aggregation* parameters.

Algorithm 3. N-To-M Aggregation for Inserts Only

Input: F - a set of flex-objects; P_G - grouping parameters; P_B - bin-packing parameters; P_A - N-to-1 aggregation parameters;
Output: F_A - a set of aggregated flex-objects;

```
1  Function AGG-N-to-M(F, P_G, P_B, P_A):
2    initAgg(P_G, P_B, P_A);
3    foreach f ∈ F do
4      processDelta(f,'+');
5    end
6    ΔF_A ← aggregateInc();
7    foreach (f, c) ∈ ΔF_A do
8      switch c do
9        case '+'
10         F_A ← F_A ∪ {f};
11       end
12     endsw
13   end
14   return F_A;
```

In Algorithm 3, *initAgg* initializes a number of global data structures (variables) for a specific setting of *grouping*, *bin-packing*, and *aggregation* parameters (Line 2). *processDelta* processes each provided delta from the sequence of deltas, in this case the sequence $(f_1, +), \ldots, (f_N, +)$ generated in Lines 3-4. *aggregateInc* returns all (latest) changes of aggregated flex-objects that occurred since its previous invocation. As it is invoked only once in Algorithm 3 (with an initially empty set of flex-objects), it returns a sequence of aggregated flex-object insert deltas (Line 6), which are then converted into a set of aggregated flex-objects F_A (Lines 7-10), returned as output (Line 14).

Additionally, the function *processDelta* can handle *delete* deltas (of the type '-') in Line 4. In this case, *aggregateInc* might return insertion (+), deletion (−), and modification (*) deltas, denoting that new flex-objects were added or existing flex-objects were deleted or changed in the set of aggregated flex-objects during the processing of deltas in Line 6. In the following, we discuss each of these functions (*initAgg*, *processDelta* and *aggregateInc*) in more detail and discuss how they together support the logical phases of *grouping*, *bin-packing*, and *N-to-1 aggregation*.

Function initAgg. This function initializes the global data structures, including *group hash*, *bin hash*, and *aggregate hash* holding groups, sub-groups (bins), and aggregates generated in the *grouping*, *bin-packing*, and *N-to-1 aggregation* phases. It also initializes the *group changes list* that stores (recent) group modifications.

Algorithm 4. The Pre-Grouping Function *ProcessDelta*

Global data: GH - a group hash; CL - group changes list;
Input: f - a flex-object to be updated; c - the type of delta: '+' for insertion; '−' for deletion;

```
1  Function processDelta(f, c):
2    p ← mapToPoint(f);
3    g ← GH.findGroup(p);
4    switch c do
5      case '+'
6        if g = NULL then
7          g ← newGroup(p);
8          GH.insertGroup(g);
9        end
10       g.insertFOs({f});
11       CL.registerChange(g,'+',{f});
12     end
13     case '−'
14       if g ≠ NULL then
15         g.removeFOs({f});
16         CL.registerChange(g,'−',{f});
17       end
18     end
19   endsw
```

Function processDelta (Algorithm 4). This function performs *pre-grouping*, i.e., it maintains flex-object groups in the group hash according to a delta (f, c) given as input. Here, f is the affected flex-object and c is the type of the delta ('+' or '−'). First, f is mapped into a d-dimensional point (Line 2). This point belongs to a cell in a d-dimensional uniform grid. The extent of a cell in each dimension is defined by the tolerance thresholds from the grouping parameters, and every cell is identified by its coordinates in the grid. Only *populated* cells with flex-objects from F are tracked, combining adjacent populated cells into a *group*. The groups are stored in the group hash, which is an in-memory hash table with the cell coordinates as the key and the flex-object group associated with this cell as the value. In Algorithm 4, the group hash is probed for an affected group (Line 3), and the respective changes are made to the group while registering *add* (+) or *delete* (−) modifications in the *group changes list* (Lines 4-16). In case a group is not found in the group hash, a new group with a single populated cell is created (Line 7). If the group changes list already contains a change record for a particular group, the record is updated to reflect the combination of the changes (Lines 11 and Line 16).

Fig. 6 shows the effect of adding a flex-object f_1 using *processDelta*. f_1 is mapped to a 2-dimensional point that lies in grid cell c_2 (Line 2). The coordinates of c_2 are used to locate a group in the group hash (Line 3). The found group is updated by inserting f_1 into its list of flex-objects (Line

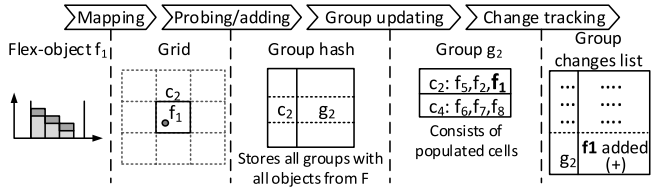


Fig. 6. Processing the addition of a flex-object in the grouping phase.

10). Finally, a change record indicating that f_1 was *added* (+) to the group is inserted into the group changes list (Line 11).

Function aggregateInc (Algorithm 5). This function finalizes *grouping* and performs (the full phases of) *bin-packing* and *N-to-1 aggregation*. First, it optimizes each of the affected groups using the *optimizeGroup* sub-function while storing new group updates in the group changes list (Line 3). Then, each group change from the updated list is processed with the sub-functions *updateBins* and *updateAggs*, which perform bin-packing and N-to-1 aggregation. Finally, the aggregated flex-object deltas are accumulated (Line 8) and returned as output (Line 13), clearing the group changes list (Line 12).

Algorithm 5. The grouping, bin-packing, and N-to-1 aggregation function *aggregateInc*

```
   Global data: CL - the group changes list;
   Output: ΔF_A - a set of aggr. flex-object deltas;
1  Function aggregateInc():
2    foreach Δgrp ∈ CL do
3      optimizeGroup(Δgrp.getGroup());
4    end
5    foreach Δgrp ∈ CL do
6      foreach Δbin ∈ updateBins(Δgrp) do
7        foreach Δf_a ∈ updateAggs(Δbin) do
8          ΔF_A = ΔF_A ∪ {Δf_a};
9        end
10     end
11   end
12   CL.clear();
13   return ΔF_A;
```

Sub-function Optimizegroup (Algorithm 6). This function is used to optimize affected groups, making them more compact and balanced. First, this function computes a minimum bounding rectangle (MBR) over all points containing flex-objects from a given group g (*getFoMBR* in Line 3). If the extent of the MBR exceeds the grouping parameter thresholds in at least one dimension (Line 3), g is split into a number of sub-groups satisfying the thresholds. For this purpose, g is first disassembled into a number of cells and then *bottom-up hierarchical clustering* [3] is performed on the MBRs of the flex-objects within these cells to form new sub-groups (Line 6). All relevant group changes are registered in the group hash and the group changes list (Lines 4-8). Additionally, group merging is considered, while first probing the group hash to identify adjacent neighbouring groups (Line 11). If the extent of the MBR computed over all flex-object points from g and an adjacent group are within the grouping parameter thresholds in all dimensions (Lines 12-13), then g and the adjacent groups are merged while tracking all associated group changes (Lines 14-18). Fig. 7 demonstrates group splitting and merging.

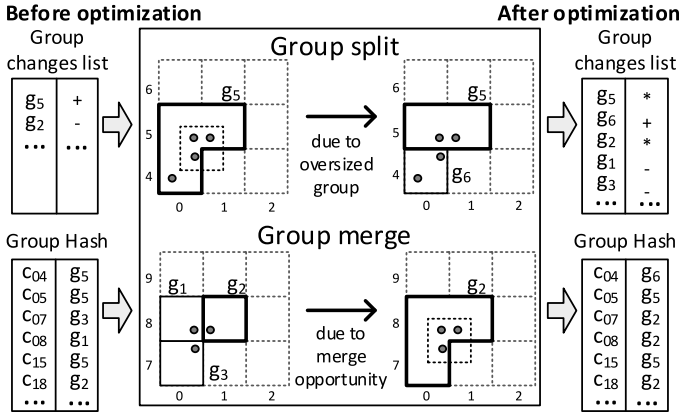


Fig. 7. Flow of data in the group optimization.

Algorithm 6. The Group Optimization Function *Optimize Group*

```
   Global data: GH - the group hash; CL - the group changes
                list; P_G - grouping parameters;
   Input: g - a group to be optimized;
1  Function optimizeGroup(g):
2    F_a ← g.getFlexObjs();
3    if getFoMBR(F_a) ≤ P_G.getMBR() then
       // Tolerances violated, split g
4      g.removeFOs(F_a);
5      CL.registerChange(g,'−', F_a);
6      foreach s ∈ clusterHierarch(g.getCells(), P_G) do
7        GH.insertGroup(s);
8        CL.registerChange(s,'+', s.getFlexObjs());
9      end
10   else          // Consider merging neighbours
11     foreach n ∈ GH.findNbrGroups(g) do
12       F_a ← g.getFlexObjs() ∪ n.getFlexObjs();
13       if getFoMBR(F_a) ≤ P_G.getMBR() then
14         F_n ← n.getFlexObjs();                  // Merge
15         n.removeFOs(F_n);
16         CL.registerChange(n,'−', F_n);
17         g.insertFOs(F_n);
18         CL.registerChange(g,'+', F_n);
19       end
20     end
21   end
```

Sub-function *updateBins* (Algorithm 7). This function propagates a given group delta Δgrp to a number of bins stored in the *bin hash*, which is a hash table with a group ID as the key. First, added and deleted flex-objects, Δ_{added} and Δ_{delete}, are retrieved (Lines 2-3), and flex-objects from Δ_{delete} are discarded from the existing bins (Lines 4-6). Groups with a total weight less than w_{min} are deleted and flex-objects from these groups and Δ_{added} are included into other existing bins using the first fit decreasing strategy [4], creating new bins with total weight less than w_{max}, if needed (Line 8). Finally, the changes of affected bins are computed and returned as output (Lines 9-12).

Sub-function Updateaggs (Algorithm 8). This function maintains the aggregate hash, which is a hash table, mapping the ID of an individual bin to an aggregated flex-object. First, a bin, bin ID, an aggregated flex-object f_a,

and the sets of added and deleted flex-objects Δ_{add} and Δ_{delete} are retrieved for an associated bin delta Δbin (Lines 2-6). Then, f_a is incrementally maintained using the N-to-1 aggregation (Algorithm 1): if there are no deletes, f_a is re-aggregated together with flex-objects from Δ_{add} (Lines 7-10); otherwise, flex-objects from the bin are aggregated into f_a from scratch (Lines 16-18), deleting aggregated flex-objects of empty bins if needed (Lines 12-14). Finally, all aggregated flex-object deltas are provided as output (Line 21).

Algorithm 7. The Bin-Packing Function *UpdateBins*

```
   Global data: BH - the bin hash;
   Input: Δgrp - a delta of a group;
   Output: ΔB - a set of bin deltas;
 1 Function updateBins(Δgrp):
 2   Δ_add ← (Δgrp).getAddedFOs();
 3   Δ_delete ← (Δgrp).getDeletedFOs();
 4   bins ← BH.getBins((Δgrp).getGrp().getId());
 5   foreach b ∈ bins do
 6     b.removeFOs(Δ_delete);
 7   end
 8   firstFitDecreasing(bins, Δ_add);
 9   ΔB ← ∅;
10   foreach b ∈ bins do
11     if b.isAffected() then
12       ΔB ← ΔB ∪ {b.getBinDelta()};
13     end
14   end
15   return ΔB;
```

The *average-case* complexity of Algorithm 3 combining all these sub-functions is $\mathcal{O}(|F| \cdot P_{avg})$, where P_{avg} is the average length of aggregated flex-object profiles. The complexity analysis details and the *worst-case* complexity are given in Appendices A.3–A.4.

4.4 M-to-N Disaggregation and Discussion

To disaggregate the instances of aggregated flex-objects, the *DAGG-1-to-N* function from Section 3 is applied independently to each of the affected instances—which are generated in a single (scheduling) step by a single entity (BRP) rather than by multiple (consumer/producer) entities. Therefore, the (M-to-N) disaggregation is straightforward in comparison to the (N-to-M) aggregation, and we therefore omit the presentation of an explicit disaggregation algorithm.

In summary, the discussed N-to-M aggregation and (M-to-N) disaggregation techniques allow efficiently aggregating flex-objects and disaggregating their instances. As the N-to-1 aggregation and the N-to-1 disaggregation functions from Section 3 are used inherently, *amount conservation* is ensured. Different grouping parameter combinations allow building flex-object groups with different levels of flex-object similarity and thus controlling the trade-off between flex-object compression and flexibility loss. The *aggregate constraint* is ensured by bin-packing. Finally, *incremental updates* are feasible, which allow efficiently processing flex-object additions and removals without the need to recompute aggregated flex-objects from scratch.

Algorithm 8. The N-to-1 Aggregation *UpdateAggs*

```
   Input: Δbin - a delta of a bin;
   Output: ΔF_a - a set of aggr. flex-object deltas
   Data: AH - the aggregate hash;
 1 Function updateAggs(Δbin):
 2   bin ← (Δbin).getBin();
 3   binId ← bin.GetId();
 4   f_a ← AH.getAgg(binId);
 5   Δ_add ← (Δbin).getAddedFOs();
 6   Δ_delete ← (Δbin).getDeletedFOs();
 7   if Δ_delete = ∅ then // No deletes
 8     f_a ← AGG-N-to-1({f_a} ∪ Δ_add);
 9     AH.updateAgg(binId, f_a);
10     ΔF_a ← ΔF_a ∪ {(f_a, '+')};
11   else // Handle deletes
12     if (bin.getFlexObjs() = ∅ then
13       AH.deleteAgg(binId);
14       ΔF_a ← ΔF_a ∪ {(f_a, '−')};
15     else // Aggregate from scratch
16       f_a ← AGG-N-to-1(bin.getFlexObjs());
17       AH.insertAgg(binId, f_a);
18       ΔF_a ← ΔF_a ∪ {(f_a, '*')};
19     end
20   end
21   return ΔF_a;
```

5 Balance Aggregation

In Section 3, we have proposed the use of *balance-alignment* to satisfy the *balance requirement*. In relation to *balance-alignment*, we now describe several extensions to the N-to-M aggregation and practical techniques for solving the (non-linear) minimization problem of finding the profile positions $p_{f_1}, p_{f_2}, \ldots, p_{f_{|F|}}$ and thus building aggregated flex-objects with the amount bounds allocated around zero (see Fig. 3b). The extensions are applicable to the N-to-1 aggregation phase (Section 4.2) and aggregate only flex-objects that are placed in the same bin.

Exhaustive search (ES). This technique explores all feasible profile position combinations of all the flex-objects in a bin. After examining all possible combinations, the technique generates an aggregated flex-object with the minimum *absolute balance* (see Section 2).

Zero terminated exhaustive search (ZES). This technique is similar to the previous one. For each bin, it stops examining further combinations when the current aggregated flex-object has an absolute balance equal to zero.

Dynamic simulated annealing (DSA). This is an approximate technique based on *simulated annealing* [5]. The parameter h is used to limit the total number of random combinations to be examined. By default, h is set to half of all possible combinations, where the total number of combinations is computed individually for each bin. The algorithm terminates if a solution with absolute balance equal to zero is found.

Simple greedy (SG). This technique, described in Algorithm 9, starts by selecting (Line 2) and removing from the bin the flex-object f_{nom} with the largest negative

balance v (Line 3); where $balance(f)$ is the sum of the average values of all the slices of a flex-object f. The technique further aggregates f_{nom} with the flex-object f_{tmp} that has (non-absolute) balance closest to $-v$ (Line 6). It examines all the profile positions of f_{tmp} and selects the aggregated flex-object f_a that gives the smallest absolute balance (Lines 7-12). In case the absolute balance of f_a is lower than the absolute balance of f_{nom} (Line 13), the algorithm continues aggregation of the same aggregated flex-object by assigning f_a to f_{nom}. Otherwise (Line 15), it stores f_{nom} as a new aggregated flex-object (Line 16) and starts a new aggregation by selecting f_{nom}, the flex-object with the largest negative balance, from the remaining flex-objects (Line 17). The technique stops when there are no more flex-objects to be aggregated (Line 4, when $|B| = 0$). As a result, the technique might produce more than one aggregated flex-object per bin (Line 20).

Algorithm 9. Simple Greedy Technique

Input: B - a bin of flex-objects;
Output: B_a - a set of aggregated flex-objects;
Data: $f_{nom}, f_a, f_{tmp}, f_x$ - flex-objects;

```
1  Function SimpleGreedy(B):
2    f_nom ←FlexObjectWithMinBalance(B);
3    B ← B \ f_nom; B_a ← ∅;
4    while |B| > 0 do
5      f_a ← f_nom; v←balance(f_nom);
6      f_tmp←ClosestTo-vBalance(B, v);
7      foreach profile position of f_tmp do
8        f_x ←AGG-N-to-1(f_tmp ∪ f_nom);
9        if AbsBalance(f_x) < AbsBalance(f_a) then
10         f_a ← f_x;
11       end
12     end
13     if AbsBalance(f_a) < AbsBalance(f_nom) then
14       B ← B \ f_tmp; f_nom ← f_a;
15     else
16       B_a ← B_a ∪ f_a; B ← B \ f_nom;
17       f_nom ←FlexObjectWithMinBalance(B);
18     end
19   end
20   return B_a
```

Exhaustive greedy (EG). Similar to simple greedy, this technique (Algorithm 10) also produces more than one aggregated flex-object per bin. It starts by selecting (Line 2) and removing (Line 3) the flex-object f_{nom} from the bin with the maximum absolute balance. However, it considers all the time flexibility values of all the remaining flex-objects (Line 6 and Line 7) in the bin to find the one that, in combination with the (intermediate) aggregated flex-object f_a, reduces the absolute balance the most. Similar to the simple greedy algorithm, it continues the aggregation of the same aggregated flex-object by assigning f_a to f_{nom} if the absolute balance is reduced (Line 10). Otherwise, it stores f_{nom} as a new aggregated flex-object (Line 17) and starts a new aggregation by selecting from the remaining flex-objects the flex-object f_{nom} having the largest absolute balance (Line 18). The technique stops when there are no more flex-objects to be aggregated.

Algorithm 10. Exhaustive Greedy Technique

Input: B - a bin of flex-objects;
Output: B_a - a set of aggregated flex-objects;
Data: $f_{nom}, f_a, f_{tmp}, f_x$ - flex-objects;

```
1  Function ExhaustiveGreedy(B):
2    f_nom ←FlexObjectWithMaxAbsBalance(B);
3    F ← F \ f_nom; F_a ← ∅;
4    while |B| > 0 do
5      f_a ← f_nom;
6      for f ∈ B do
7        for all profile positions of f do
8          f_x ←AGG-N-to1(f ∪ f_nom);
9          if AbsBalance(f_x) < AbsBalance(f_a)
           then
10           f_a ← f_x; f_tmp ← f;
11         end
12       end
13     end
14     if AbsBalance(f_a) < AbsBalance(f_nom) then
15       B ← B \ f_tmp; f_nom ← f_a;
16     else
17       B_a ← B_a ∪ f_a; B ← B \ f_nom;
18       f_nom←FlexObjectWithMaxAbsBalance(B);
19     end
20   end
21   return B_a
```

6 Experimental Evaluation

In this section, we present the evaluation of the N-to-M aggregation and the balance aggregation techniques.

6.1 Experimental Setup

We implemented the basic N-to-M and balance aggregation techniques in Java 1.6. As there are no other flex-object aggregation and disaggregation solutions, we compared our N-to-M aggregation implementation to two rival implementations: R_{HA} and R_{SG}, using our solution for bin-packing and N-to-1 aggregation, but with different (non-incremental) implementations for the grouping phase.

- For grouping, R_{HA} uses agglomerative hierarchical clustering [3] by first assigning each flex-object to individual clusters and then repeatedly merging the two closest clusters while no grouping constraints are violated. The distance between two clusters is calculated based on the values of the grouping parameter flex-object attributes.
- R_{SG} applies the *similarity group-by operator* [6] for one grouping parameter at a time, thus partitioning the input into valid groups of similar flex-objects.

Our experiments were run on a PC with Quad Core Intel Xeon E5320 CPU, 16 GB RAM, OpenSUSE 11.4 (x86_64).

6.2 N-to-M Aggregation

We evaluated the N-to-M aggregation discussed in this paper as well as R_{HA} and R_{SG} using a synthetic flex-object dataset from the MIRABEL project, containing one million flex-objects representing consumption. The *earliest start time* (t_{es}) is distributed uniformly in the range $[0, 23228]$. The

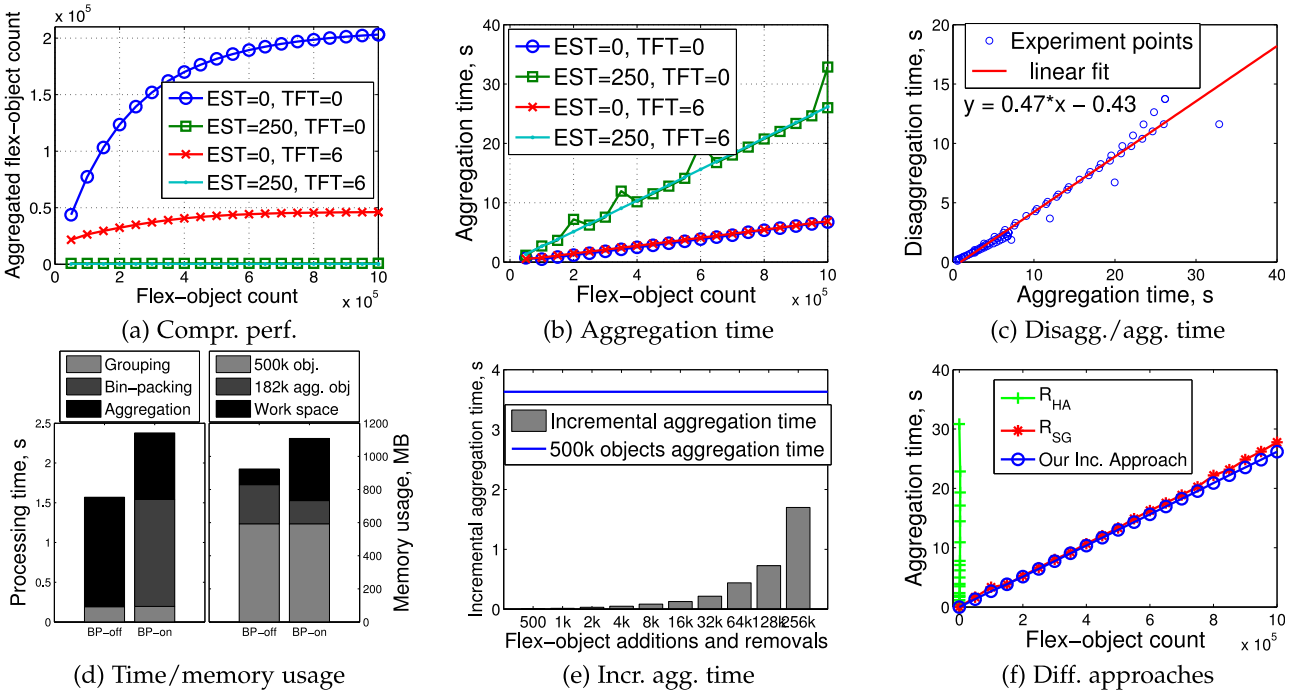


Fig. 8. Scalability and incremental behaviour.

number of slices and the time flexibility values ($t_{ls} - t_{es}$) follow the normal distributions $\mathcal{N}(8, 4)$ and $\mathcal{N}(20, 10)$ in the ranges $[10, 30]$ and $[4, 12]$; the slice duration is fixed to 1 time unit (15 minutes) for all flex-objects, thus length of the profiles ranges from 2.5 to 7.5 hours. Unless otherwise stated, the default values of the experimental parameters are: (a) the number of flex-objects is $500\,k$; (b) $EST = 0$ (*Earliest Start Time Tolerance*) and $TFT = 0$ (*Time Flexibility Tolerance*) are used as the grouping parameters. They are based on the *Earliest Start Time* (t_{es}) and *Time Flexibility* ($t_{ls} - t_{es}$) flex-object attributes. (c) the aggregate constraint is unset (bin-packing is disabled). We also perform experiments with bin-packing enabled (explicitly stated).

Scalability. To evaluate flex-object compression in terms of performance and scalability, the number of flex-objects is gradually increased from 50 to $1000\,k$. Aggregation is performed using two different EST and TFT parameter values: EST equal to 0 or 250, and TFT equal to 0 or 6. Disaggregation is executed with randomly generated instances of aggregated flex-objects. The results are shown in Figs. 8a and 8d. Figs. 8a and 8b show that different aggregation parameter values lead to different compression factors and aggregation times. Disaggregation is approx. two times faster than aggregation (see Fig. 8c) regardless of the flex-object count and grouping parameter values. Most of the time is spent on the bin-packing (if enabled) and N-to-1 aggregation phases (the two left bars in Fig. 8). Considering the overhead associated with incremental behaviour, the amount of memory used by the approach is relatively small compared to the footprint of the original and aggregated flex-objects. Memory usage increases when bin-packing is enabled.

Incremental Behaviour. To evaluate incremental aggregation, we first aggregate $500\,k$ flex-objects. Then, for different k values ranging from 500 to $256\,k$, we insert k new flex-objects and remove k randomly selected flex-objects. The total number of flex-objects stays at $500k$. For every value of k, we execute incremental aggregation. Fig. 8e shows that updates can be processed efficiently. Hence, our approach results in substantial time savings compared to the case when all $500\,k$ flex-objects are aggregated from scratch (represented by the line in Fig. 8e). We then compare the total time to process flex-objects with our incremental approach to the other two (inherently non-incremental) approaches (R_{HA} and R_{SG}). As Fig. 8f illustrates, our approach is competitive in comparison to R_{SG} in terms of scalability. The overhead associated with the change tracking and group optimization in the incremental grouping phase is not significant in the overall aggregation time. Additionally, the hierarchical clustering-based approach (R_{HA}) results in very high execution time even for small datasets (due to a large amount of distance computations) and is thus not scalable enough for flex-object aggregation.

Effect of grouping parameters. As shown in Fig. 9a, EST significantly affects the flex-object compression factor. For this dataset, increasing EST by a factor of two leads to a flex-object reduction by approximately the same factor. However, the use of high EST values results in aggregated flex-object profiles with more slices. Aggregating these requires more time (see "aggregation time" in Fig. 9a). The TFT parameter has a significant impact on the flexibility loss (see "flexibility loss" in Fig. 9b). Higher values of TFT results in higher flexibility losses. When TFT is set to 0, aggregation causes no flexibility loss but results in a larger number of aggregated flex-objects. When the number of distinct time flexibility values in a flex-object dataset is low (as in our case), the best compression with no flexibility loss can be achieved when $TFT = 0$ and the other grouping parameters are unset (or set to high values).

Optimization and bin-packing. We now evaluate group optimization and bin-packing. As shown in Figs. 9c and 9d, group optimization is relatively cheap (Fig. 9d) and it

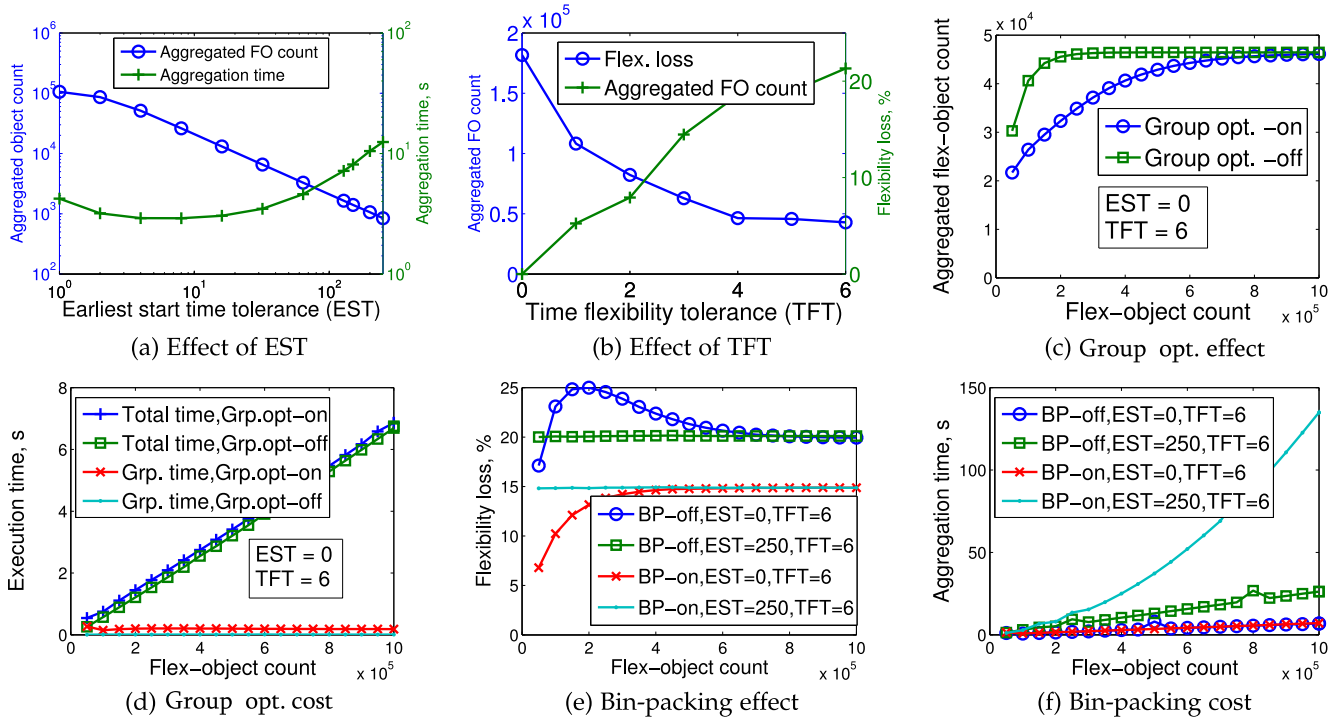


Fig. 9. Grouping, optimization, and bin-packing.

substantially contributes to the reduction of the number of aggregated flex-object (Fig. 9c). To evaluate bin-packing, the aggregate constraint was set so that the time flexibility of an aggregate is always at least 8 ($w_{min} = 8$, equiv. to 2 hours). By enabling this constraint, we investigate the overhead associated with bin-packing and its effect on the flexibility loss. As shown in Fig. 9e, by restricting the time flexibility for every aggregate, the overall flexibility loss can be limited. However, bin-packing introduces a substantial overhead that depends on the number of objects in flex-object groups after the group optimization (see Fig. 9f). When this number is small ($EST = 0$, $TFT = 6$), the overhead of bin-packing is insignificant. However, when groups are large ($EST = 250$, $TFT = 6$), bin-packing overhead becomes very significant.

6.3 Balance Aggregation

We experimentally evaluated the balance aggregation techniques, namely exhaustive search, zero terminated exhaustive search, dynamic simulated annealing, simple greedy, and exhaustive greedy. We compared these techniques against each other, as well as to start-alignment, in terms of *absolute balance*, *flexibility loss*, and *execution time* by varying the grouping tolerances *EST* and *TFT*. We also evaluate the techniques in terms of grouping parameters. We set *EST* to zero and test values from zero to six for the *TFT* parameter using datasets with 40 customers, which contain approximately 90K flex-objects (Figs. 10c, 11c, and 11f. We used eight distinct flex-object datasets derived from the MIRABEL dataset. The first dataset is generated from the historical measurements of five randomly selected customers (households), the second by adding another five customers, and so on, up to the eighth, which contains 40 customers in total.

We run all experiments ten times using ten different instances of the described eight datasets. The average results of the experiments are illustrated in Figs. 10 and 11c and Table 1. In addition, we also show the full variation of execution times in Figs. 11d and 11f) since their variations were much higher than the variations of flexibility loss and absolute balance. We often omitted the results of ES, ZES, and DSA due to the extremely high execution times when $TFT > 0$. We also omitted DSA in Fig. 11a since

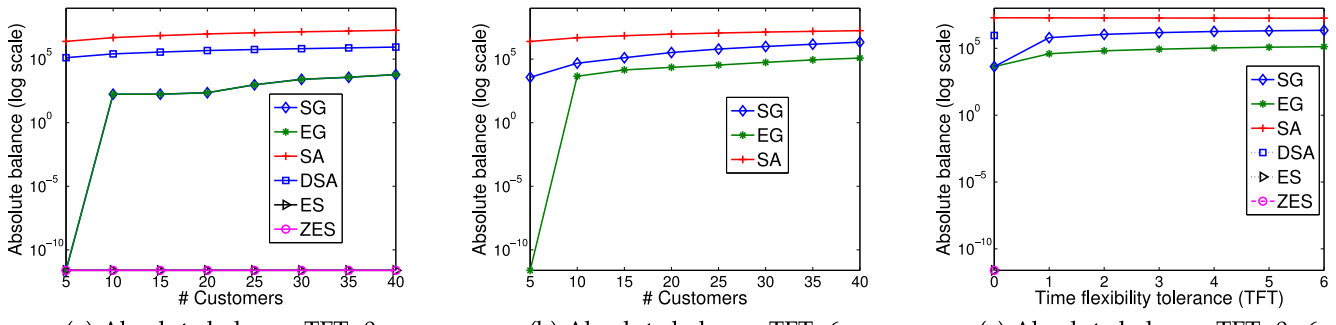
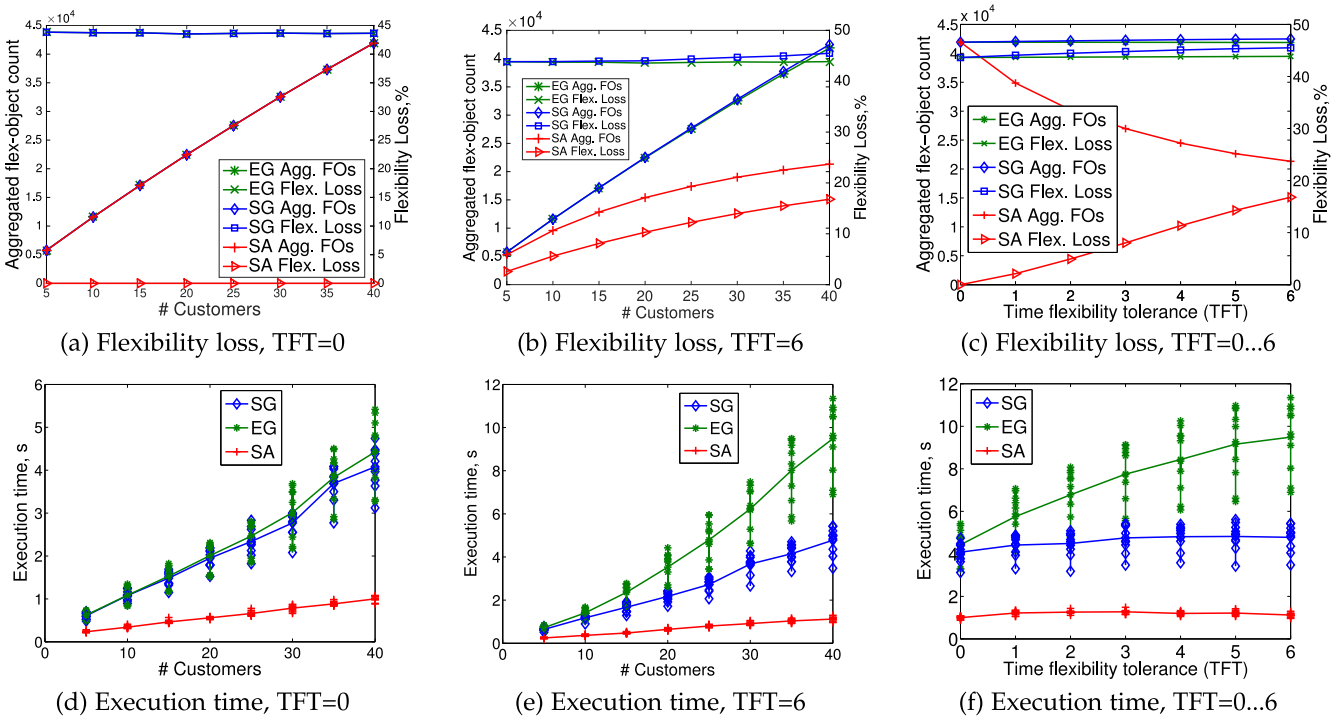


Fig. 10. Absolute balance.



(a) Flexibility loss, TFT=0 (b) Flexibility loss, TFT=6 (c) Flexibility loss, TFT=0...6

(d) Execution time, TFT=0 (e) Execution time, TFT=6 (f) Execution time, TFT=0...6

Fig. 11. Flexibility loss and execution time.

it shows the highest flexibility loss due to the limited number of iterations.

Absolute balance. Both ES and ZES achieve zero absolute balance, which is possible due to the nature of the test data, see Fig. 10a. In addition, EG and SG achieve a very low absolute balance compared to DSA and SA, see Figs. 10a and 10c. SA obtains the highest absolute balance in all examined scenarios since it does not consider balancing during aggregation.

Flexibility loss. SA has the least flexibility loss in all the scenarios followed by EG and SG (Figs. 11a and 11c). ES and ZES have similar percentages of flexibility loss compared to EG and SG; hence we omit them in the figures. This happens because their goal is to achieve the minimum absolute balance and the flex-objects participating in the aggregation are time shifted in a similar way. In Fig. 11b, we see that the flexibility loss of SA increases when more flex-objects participate in the aggregation and the number of aggregated flex-objects follows the same behaviour. This happens because when $TFT > 0$, the flex-objects in each group have different latest start times and therefore the aggregated flex-object will have smaller time flexibility intervals. In Figs. 11b and 11c, we see that both EG and SG create more aggregated flex-objects than SA due to their capability to generate more than one aggregated flex-object for each bin.

Execution time. SA is the fastest among all the techniques, followed by SG and EG, see Figs. 11d and 11f. SA's execution time is correlated with the number of flex-objects and, since this is constant, execution time shows a similar behaviour, see Fig. 11f. On the other hand, as *TFT* grows, SG is not examining a larger solution space—in contrast to EG—and generates approximately the same number of aggregated flex-objects, see Figs. 11c and 11f. Therefore, the execution time remains almost constant.

TABLE 1
Execution Time (15 customers, $EST = 0$, $TFT = 5$)

Technique	Execution time
Start alignment	0.5 sec.
Simple greedy	1.7 sec.
Exhaustive greedy	2.3 sec.
Dynamic simulated annealing	26 min.
Zero terminated exhaustive search	14.27 hours
Exhaustive search	18.76 hours

6.4 Experiment Summary

In summary, we show that our basic N-to-M aggregation technique scales linearly with the number of flex-object inserts, and it is just as efficient as the best non-incremental grouping approach (R_{SG}). The overhead associated with the incremental behaviour is insignificant. The trade-off between the flex-object compression factor and flexibility loss can be controlled using the grouping parameters. The compression factor can be further increased efficiently by group optimization. Disaggregation is approx. two times faster than aggregation.

Furthermore, we show that the balance aggregation techniques EG and SG can achieve a very low absolute balance with low execution time and maintain a near-constant behaviour regarding flexibility loss. For these characteristics, EG performs better than SG and produces a lower number of aggregated flex-objects. However, all the proposed balance aggregation techniques show higher flexibility loss compared to SA, thus demonstrating the inevitable trade-off between flexibility loss and absolute balance. For larger grouping tolerance values, the difference in flexibility loss between SA and EG/SG is reduced.

7 Related Work

Clustering. Many clustering algorithms have been proposed, including density-based (e.g., BIRCH [7]), centroid-based (e.g., K-Means [8]), hierarchical clustering (e.g., SLINK [3], [9]) as well as incremental algorithms, such as incremental K-means [10] and incremental BIRCH [11]. In comparison to our approach, clustering partially solves only the grouping part, which is substantially simpler than the whole problem of grouping, bin-packing, balance-aligning, and N-to-1 aggregation. For grouping alone, existing clustering algorithms provide neither effective incremental solutions nor strict guaranties of cluster object similarity, both of which are provided by our approach. The closest work is incremental grid-based clustering [12], [13], [14], where we, in comparison, improve the clusters across the grid boundaries and limit the number of items per cluster using bin-packing.

Similarity group by. The operator SGB [15] groups objects based on the similarity between tuple values and is supported by SimDB [15]. However, SGB again solves only the grouping part of the problem and is (unlike our approach) not incremental, which is essential.

Complex objects. Complex objects with multidimensional data exist in many real-world applications [16] and can be represented with *multidimensional data models* [17]. Several research efforts (e.g., [18] and [19]) have been proposed to aggregate complex objects. However, these efforts do not consider the specific challenges related to aggregating flex-objects.

Temporal aggregation. Several papers have addressed aggregation for temporal and spatio-temporal data including instantaneous temporal aggregation [20], cumulative temporal aggregation [21], [22], [23], histogram-based aggregation [24], and multi-dimensional temporal aggregation [25]. These techniques differ in how a time line is partitioned into time intervals and how an aggregation group is associated with each time instant. The efficient computation of these time intervals poses a great challenge and therefore various techniques that allow computing them efficiently have been proposed [26], [27], [28]. Unfortunately, these techniques only deal with simple data items without flexibilities, making them unsuitable for aggregation of flex-objects.

Balance aggregation. Existing approaches [29] solve the energy balancing problem globally and independently from the aggregation. In comparison, our approach performs local balancing during aggregation, thus reducing distribution grid congestion risks and partially fulfilling the goal of scheduling (global balancing).

8 Conclusion and Future Work

Objects with inherent flexibilities in *time* and *amount*, so-called flexibility objects (flex-objects), occur in both scientific and commercial domains. By focusing on flex-object aggregation and disaggregation, this paper formally defined relevant concepts and provided a novel and efficient grid-based incremental technique for flex-object aggregation and disaggregation. The paper considered the aggregation/disaggregation process, the grouping of flex-objects, alternatives for computing aggregates, and associated requirements. Additionally, it introduced the concept of energy balancing in aggregation and proposed five techniques that fulfil both balancing and other aggregation goals simultaneously. Extensive experiments using data from an energy domain project showed that the techniques provide very good performance while satisfying all entailed requirements.

Future work will address challenges of aggregating and disaggregating flex-objects with additional flexibilities in profile slice duration. Furthermore, novel balance aggregation techniques taking into account electricity grid constraints will be examined, together with advanced grouping techniques and data structures to further improve performance.

Acknowledgments

This work was supported in part by the TotalFlex project sponsored by the ForskEL program of Energinet.dk.

References

[1] M. Boehm, L. Dannecker, A. Doms, E. Dovgan, B. Filipic, U. Fischer, W. Lehner, T. B. Pedersen, Y. Pitarch, L. Siksnys, and T. Tusar, "Data management in the MIRABEL smart grid system," in *Proc. 1st Workshop Energy Data Manage.*, 2012, pp. 95–102.

[2] L. Šikšnys, M. E. Khalefa, and T. B. Pedersen, "Aggregating and disaggregating flexibility objects," in *Proc. 24th Int. Conf. Sci. Statist. Database Manage.*, 2012, pp. 379–396.

[3] W. H. Day and H. Edelsbrunner, "Efficient algorithms for agglomerative hierarchical clustering methods," *J. Classification*, vol. 1, no. 1, pp. 7–24, Dec. 1984.

[4] R. E. Korf, "A new algorithm for optimal bin packing," in *Proc. 18th Nat. Conf. Artif. Intell.*, 2002, pp. 731–736.

[5] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, 1st ed. New York, NY, USA: Wiley, 1988.

[6] Y. N. Silva, W. G. Aref, and M. H. Ali, "Similarity group-by," in *Proc. Int. Conf. Data Eng.*, 2009, pp. 904–915.

[7] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1996, pp. 103–114.

[8] J. B. Macqueen, "Some methods of classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Stat. Prob.*, 1967, pp. 281–297.

[9] R. Sibson, "SLINK: An optimally efficient algorithm for the single-link cluster method," *Comput. J.*, vol. 16, no. 1, pp. 30–34, Jan. 1973.

[10] Z. Zhang, Y. Yang, A. K. H. Tung, and D. Papadias, "Continuous k-means monitoring over moving objects," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 9, pp. 1205–1216, Sep. 2008.

[11] C. S. Jensen, D. Lin, and B. C. Ooi, "Continuous clustering of moving objects," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 9, pp. 1161–1174, Sep. 2007.

[12] N. H. Park and W. S. Lee, "Statistical grid-based clustering over data streams," *SIGMOD Rec.*, vol. 33, no. 1, pp. 32–37, 2004.

[13] G. Hou, R. Yao, J. Ren, and C. Hu, "A clustering algorithm based on matrix over high dimensional data stream," in *Proc. Int. Conf. Web Inf. Syst. Mining*, 2010, pp. 86–94.

[14] G. Lei, X. Yu, X. Yang, and S. Chen, "An incremental clustering algorithm based on grid," in *Proc. 8th Int. Conf. Fuzzy Syst. Knowl. Discovery*, 2011, pp. 1099–1103.

[15] Y. N. Silva, A. M. Aly, W. G. Aref, and P.-A. Larson, "SimDB: A similarity-aware database system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 1243–1246.

[16] E. Malinowski and E. Zimányi, *Advanced Data Warehouse Design: From Conventional to Spatial and Temporal Applications*, 1st ed. New York, NY, USA: Springer, 2008.

[17] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson, "A foundation for capturing and querying complex multidimensional data," *Inf. Syst.*, vol. 26, no. 5, pp. 383–423, 2001.

[18] J. Cabot, J.-N. Mazón, J. Pardillo, and J. Trujillo, "Specifying aggregation functions in multidimensional models with OCL," in *Proc. 29th Int. Conf. Conceptual Model.*, 2010, pp. 419–432.

[19] D. Zhang, "Aggregation computation over complex objects," Ph.D. dissertation, Univ. California, Riverside, CA, USA, 2002.
[20] M. H. Böhlen, J. Gamper, and C. S. Jensen, "How would you like to aggregate your temporal data?" in *Proc. IEEE 13th Int. Symp. Temporal Representation Reasoning*, 2006, pp. 121–136.
[21] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," *Int. J. Very Large Data Bases*, vol. 12, no. 3, pp. 262–283, 2003.
[22] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *Proc. 13th Int. Conf. Very Large Data Bases*, 2004, pp. 336–347.
[23] C. Jin and J. G. Carbonell, "Incremental aggregation on multiple continuous queries," in *Proc. IEEE 16th Int. Symp. Found. Intell. Syst.*, 2006, pp. 167–177.
[24] C.-Y. Chow, M. F. Mokbel, and T. He, "Aggregate location monitoring for wireless sensor networks: A histogram-based approach," in *Proc. 10th Int. Conf. Mobile Data Manage.: Syst., Services Middleware*, 2009, pp. 82–91.
[25] M. H. Böhlen, J. Gamper, and C. S. Jensen, "Multi-dimensional aggregation for temporal data," in *Proc. 12th Int. Conf. Adv. Database Technol.*, 2006, pp. 257–275.
[26] J. Gordevičius, J. Gamper, and M. Böhlen, "Parsimonious temporal aggregation," in *Proc. Int. Conf. Adv. Database Technol.*, 2009, pp. 1006–1017.
[27] D. Gao, J. A. G. Gendrano, B. Moon, R. T. Snodgrass, M. Park, B. C. Huang, and J. M. Rodrigue, "Main memory-based algorithms for efficient parallel aggregation for temporal databases," *Distrib. Parallel Databases*, vol. 16, no. 2, pp. 123–163, 2004.
[28] B. Moon, I. Fernando Vega Lopez, and V. Immanuel, "Efficient algorithms for large-scale temporal aggregation," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 744–759, May/Jun. 2003.
[29] T. Tusar, E. Dovgan, and B. Filipic, "Evolutionary scheduling of flexible offers for balancing electricity supply and demand," in *Proc. IEEE Congr. Evol. Comput.*, 2012, pp. 1–8.

Laurynas Šikšnys received the bachelor's degree in informatics from Kaunas University of Technology, Lithuania, in 2007. He received the MSc degree in computer engineering in 2009, and the PhD degree in computer science from Aalborg University and Technische Universität Dresden in 2014. His research interests include real-time data management architectures, energy data management, and prescriptive analytics in cyber-physical systems.

Emmanouil Valsomatzis received the Dipl.Ing. degree from the Computer Engineering and Informatics Department, University of Patras, and the MSc degree in information systems from Aristotle University, Greece, in 2010 and 2012, respectively. He is currently working toward the PhD degree in the Computer Science Department, Aalborg University, Denmark. His research addresses business intelligence solutions for Smart Grid data management. He is a student member of the IEEE.

Katja Hose is an assistant professor of computer science at Aalborg University. Before joining Aalborg University, she was a postdoctoral researcher at the Max-Planck Institute for Informatics in Saarbrücken, Germany, and received the doctoral degree in computer science from Ilmenau University of Technology. Her research interests include energy data management, Linked Data, query processing and optimization, and distributed systems.

Torben Bach Pedersen is a professor of computer science at Aalborg University, Denmark. His research concerns business intelligence and big data, especially "Big Multidimensional Data"—the integration and analysis of large amounts of complex and highly dynamic multidimensional data. He is an ACM Distinguished scientist, a senior member of the IEEE, and a member of the Danish Academy of Technical Sciences.