

Utveckling av standardiserat verktyg för visualisering och analys av sensordata

med Azure Databricks

William Kumpu

Examensarbete för ingenjör (YH)-examen

Utbildningsprogrammet för el- och automationsteknik

Vasa 2024

EXAMENSARBETE

Författare: William Kumpu
Utbildning och ort: EI- och automationsteknik, Vasa
Inriktning: Informationsteknik
Handledare: Ray Pörn, Yrkeshögskolan Novia
Anton Fagerström, Mirka

Titel: Utveckling av standardiserat verktyg för visualisering och analys av sensordata – med Azure Databricks

Datum: 30.5.2024 Sidantal: 35

Abstrakt

Detta examensarbete handlar om visualisering och hantering av data inom Azure Databricks. Uppdraget var att underlätta användningen av programmet för operatörer samt analytiker som eventuellt inte är bekanta med programmering sedan tidigare. Intervjuer utfördes tillsammans med personalen för att få fram vilken data de var intresserade av samt för att få en förståelse för hur de har hämtat och behandlat data tidigare. Sedan gjordes verktyg anpassade för deras önskemål. Det gjordes två olika verktyg för personalen.

Dessa verktyg gjordes i Azure Databricks, med hjälp av Python och SQL samt flera moduler och bibliotek som hör till, såsom PySpark. Först gjordes en notebook var användaren väljer maskin samt typ av sensor, och senare specifik sensor med tidsfiltrering. Detta verktyg skulle underlätta datasökningsprocessen för personalen och fungera som en bra startpunkt för nya användare, eftersom programmet gjordes med användarvänlighet i beaktande. Sedan gjordes en instrumentpanel för energiförbrukning per maskin med tidsfiltrering som analytiker kan använda för att se var fabriken förbrukar energi.

Examensarbetet resulterade i två lättförståeliga verktyg som personalen kan använda i Databricks, var de kan få den data de är ute efter smidigt och enkelt. Widgets gör det betydligt enklare för användaren att lägga till filtreringar, i stället för att användarna själv måste skriva eller ändra kod i SQL eller Python. Med verktygen kan personalen bli mera bekanta med Databricks och dess möjligheter, samt lätt hitta data om sensorer för analysering.

Språk: svenska

Nyckelord: notebook, instrumentpanel, SQL-sats, widget

BACHELOR'S THESIS

Author: William Kumpu
Degree Programme: Electrical Engineering and Automation, Vaasa
Specialisation: Information Technology
Supervisors: Ray Pörn, Novia University of Applied Sciences
Anton Fagerström, Mirka

Title: Development of Standardised Tools for Visualisation and Analysis of Sensor Data – With Azure Databricks

Date: 30.5.2024 Number of pages: 35

Abstract

This thesis revolves around the visualisation and management of data within Azure Databricks. The task was to facilitate the use of the program for operators and analysts who might not be familiar with programming. The mission was to conduct interviews with the staff to determine the data they are interested in and how they have previously obtained it. Then, tools tailored to their requirements were developed. Two different tools were created for the staff.

These tools were built using Azure Databricks notebooks and dashboards, utilising Python and SQL, along with several modules and libraries such as PySpark. Firstly, a notebook was created where the user could select the machine and type of sensor, and later a specific sensor with a time filter. This tool aimed to streamline the data retrieval process for the staff and serve as a good starting point for new users, given the program's high user-friendliness. Secondly, a dashboard was developed to monitor energy consumption per machine with a time filter, which analysts could use to identify which machine is using energy and when.

The outcome was two user-friendly tools that staff can use in Databricks to smoothly obtain the data they are looking for. With the help of widgets, adding filters becomes significantly easier than users having to write or modify code in SQL or Python themselves. With these tools, staff can delve deeper into the Databricks program and easily find sensor data for analysis.

Language: Swedish

Key words: Notebooks, Dashboards, SQL, Query, Database, Widget

Innehållsförteckning

1	Inledning.....	1
1.1	Uppdragsgivare	1
1.2	Bakgrund och uppdrag	2
2	Tekniker och teoretisk bakgrund	3
2.1	Azure Databricks.....	3
2.1.1	Olika sätt att lagra data	3
2.1.2	Instrumentpaneler	3
2.1.3	Notebooks	4
2.2	Python.....	4
2.3	Structured Query Language.....	5
2.4	Apache Spark	5
2.4.1	PySpark.....	6
3	Utförande.....	7
3.1	Förberedande arbete	7
3.2	Intervjuer	8
3.2.1	Frågor till respondenterna.....	8
3.2.2	Intervju A	8
3.2.3	Intervju B	8
3.2.4	Intervju C	9
3.2.5	Intervju D	9
3.2.6	Summering av intervjuer	9
3.3	Instrumentpanel över energiförbrukning	10
3.3.1	Plan för verktyget	10
3.3.2	Datauppsättningar och SQL-satser	10
3.3.3	Visualisering.....	12
3.4	Notebook för maskinsensorer	15
3.4.1	Plan för verktyget	15
3.4.2	Databricks-widgetar	15
3.4.3	Koden.....	17
4	Resultat och utvecklingsmöjligheter	26
4.1	Resultat.....	26
4.2	Utvecklingsmöjligheter för verktygen.....	30
5	Diskussion	31
6	Källförteckning	33

Förteckning av kodexempel

Kodexempel 1. SQL-sats för energiförbrukning del 1.....	10
Kodexempel 2. SQL-sats för energiförbrukning del 2.....	11
Kodexempel 3. Inläsning av widgets	17
Kodexempel 4. Definiering av dataram som lista	18
Kodexempel 5. Konvertering från användarens input till sensorernas namn	18
Kodexempel 6. Formatering av maskin-ID för SQL-sats	19
Kodexempel 7. SQL-sats för notebook med behandling av tidsintervall	19
Kodexempel 8. SQL-sats för notebook med behandling av rådata	21
Kodexempel 9. Widget-initialisering från kod	21
Kodexempel 10. Definiering av dataram för widget	21
Kodexempel 11. Kod som initialiserar widgeten för specifik sensor	21
Kodexempel 12. Kod som visar alla visualiseringar	22
Kodexempel 13. SQL-sats för visualisering av specifik sensor	23
Kodexempel 14. Fullständig SQL-sats över instrumentpanel.....	27
Kodexempel 15. Fullständig kod från första cellen ur notebooken.....	28
Kodexempel 16. Fullständig kod från den andra cellen ur notebooken.....	29

Figurförteckning

Figur 1. Figur över datatabell.	7
Figur 2. Filtrering för val av datum.....	12
Figur 3. Val av maskin id vid filtrering.	13
Figur 4. Instrumentpanel över energiförbrukning från olika maskiner.	14
Figur 5. Konfiguration av visualisering för instrumentpanel.....	14
Figur 6. Widgetar för notebook del 1.....	16
Figur 7. Widgetar för notebook del 2.....	17
Figur 8. Exempelvisualisering över hastighetssensorer över en viss tidsperiod.....	23
Figur 9. Exempelvisualisering över specifik hastighetssensor.....	24
Figur 10. Visualiseringsredigerare för notebook.....	25
Figur 11. Slutgiltig bild över instrumentpanel.	27
Figur 12. Bild av notebook widgets samt dokumentationscell.	28

Ordlista

API	Står för Application Programming Interface och tillåter olika program att överföra data mellan varandra.
Cell	En del av en notebook, en notebook består av flera celler, cellerna kan köras individuellt.
Dashboard	På svenska instrumentpanel. Inom Databricks verktyg för visualiseringar av olika data
Notebook	En interaktiv digital miljö där användarna kan kombinera kod med text och visa resultat smidigt.
Python	Ett programmeringsspråk på hög nivå som är lätt att tolka och har enkla syntax.
SQL	Står för Structured Query Language, programmeringsspråk som används för datahantering i relationella databaser.
Widget	Ett sätt för användaren att filtrera data och kod, vanligast i formen av en rullgardinsmeny, var användaren väljer olika värden.

1 Inledning

Den digitala utvecklingen inom företag är i full gång och det blir allt vanligare att olika program och mjukvarulösningar integreras i verksamheten över tid. Det är viktigt för företag att noggrant välja de program som bäst stöder deras behov. Att satsa på ett program i stället för att sprida resurser på flera alternativ som åstadkommer liknande resultat innebär inte bara en fördelaktig kostnadsbesparing, utan minskar även behovet av omfattande utbildning för personalen. Outnyttjat data kan leda till missade möjligheter för förbättring. Genom att analysera och dra nytta av data kan företag minska på förbrukning samt öka produktiviteten.

Detta arbete utfördes under våren 2024 på uppdrag av företaget Mirka. Mirka var i behov av verktyg inom Azure Databricks som personalen kan använda, i form av instrumentpaneler och notebooks. Det huvudsakliga syftet med arbetet är att underlätta datasökningsprocessen och datahantering för operatörer samt analytiker som kanske inte arbetar inom IT eller har kunskap kring databaser eller programmering. Målet med arbetet är att sänka tröskeln för operatörer samt analytiker att använda och förstå dessa Azure Databricks instrumentpaneler och notebooks.

I detta examensarbete har känsligt data dolts. Det har använts generativa AI-metoder som stöd och hjälpmedel för att förbättra texten (OpenAI, 2024).

1.1 Uppdragsgivare

Mirka Ab grundades år 1943 och är en finsk tillverkare av olika slipprodukter samt slipsystem. I början av 1960-talet hade Mirka en stor export till flera länder och för att ha sin plats i den internationella marknaden satsade Mirka på produktutveckling. Större utrymmen krävdes för att tillverka de nya produkterna och år 1972 investerade de i en modern fabrik som blev färdig året efter. År 1977 splittrades en del av produktionen till Oravais. Mirka utvidgade sig mycket internationell mellan 1972–2010, flera dotterbolag, exportkontor, samt agenturer bildades runt om i världen. I dagens läge är Mirka en del av KWH-koncernen och Mirka är världsledande i ytbehandlingsteknik. (Mirka, 2024).

1.2 Bakgrund och uppdrag

Vid Mirkas produktionsanläggning strömmas sensordata kontinuerligt upp till ett Azure Databricks Lakehouse. Data blev tillgänglig i slutet av november 2023, men används för närvarande inte till sin fulla potential. Målet med arbetet är att skapa en samling standardiserade notebooks och instrumentpaneler som möjliggör ett effektivare sätt för operatörer och analytiker att analysera, söka och visualisera data. För att skapa ändamålsenliga verktyg så utfördes först intervjuer med några anställda och sedan skapades verktygen utgående från de behov och önskemål som framkommit.

2 Tekniker och teoretisk bakgrund

Detta kapitel behandlar de olika tekniker som används i examensarbetet samt teorin bakom dem.

2.1 Azure Databricks

Azure Databricks är en del av Microsofts Azure molnplattform. Databricks är en öppen analysplattform gjord för att bygga, distribuera och underhålla olika datalösningar, analysverktyg samt AI-lösningar i stor skala. Databricks erbjuder också säker molnlagring och hanterar molninfrastruktur. Med hjälp av AI med ett datasjöhus kan Databricks förstå unikt data, den håller prestandan optimal samt hanterar infrastruktur så att den matchar användarens behov. Databricks erbjuder olika typer av verktyg för bearbetning av data samt ett enhetligt gränssnitt för de flesta datauppgifter, till exempel schemaläggning, visualisering, instrumentpaneler, hantering av säkerhet, maskininlärning och generativa AI-lösningar. (Microsoft, 2024c).

2.1.1 Olika sätt att lagra data

Ett datasjöhus, datasjö och datalager är olika typer av datahanteringssystem. Datalager har funnits runt 30 år och har använts ofta inom affärsintelligens. Datalagret är byggt för data som troligtvis inte kommer att ändras på särskilt ofta och det kan ta från några minuter till flera timmar att få vissa resultat. Datalagret bygger upp data prydligt och väl-strukturerat. Datasjön är mera modern än datalagret och anses som en motsats som används för datavetenskap och maskininlärning. Datasjön lagrar och bearbetar data billigt och effektivt men permanent. Ett datasjöhus är däremot en kombination av både datasjöns samt datalagrets goda egenskaper och erbjuder bland annat på låg fördröjning för analys samt direkt åtkomst till data vilket gör det lämpligt för affärsintelligens, datavetenskap samt maskininlärning. (Databricks, 2024e).

2.1.2 Instrumentpaneler

En instrumentpanel, även kallad "Dashboard", är en uppbyggnad av en eller flera olika visualiseringar i ett gridsystem som användaren konstruerar. En instrumentpanel består av en eller flera datauppsättningar. En datauppsättning är antingen en datatabell som

användaren har valt eller en SQL-sats som användaren gjort. Fördelen med instrumentpaneler är att de är enkla att konfigurera och lättförståeliga. Nackdelen är att när man hämtar data så kan man inte läsa in små delar före man visualiserar data, vilket kan leda till hög belastning på systemet, vilket i sin tur leder till en osmidig upplevelse. (Microsoft, 2024a).

2.1.3 Notebooks

Notebooks är ett smidigt sätt att utveckla kod och presentera resultat inom datavetenskap och maskininlärning. En notebook består av en eller flera celler, som innehåller text eller kod och varje cell kan köras skilt. Notebooks är det primära verktyget i Databricks för att skapa arbetsflöden samt samarbeta med kollegor. I notebooks kan man utveckla kod med Python, SQL, Scala och R. Man kan även skapa schemalagda arbeten för automatisering av uppgifter, såsom arbetsflöden med flera notebooks och visualiseringar som körs regelbundet. I notebooks cellsystem har man möjligheten att hämta variabler från andra celler. Databricks notebooks erbjuder även på ett Git-baserat arkiv för lagring av notebooks och filer. (Microsoft 2024b).

Ett annat känt notebooks-program är Jupyter, vilket Databricks stöder. Jupyter är mera anpassat för ensamt arbete samt behandling av mindre data. När flera arbetar på samma projekt via Jupyter så kan det bli svårt att hålla projektet i synk. Jupyter erbjuder inga inbyggda verktyg eller inbyggda datavisualiseringar på samma sätt som Databricks gör. (Databricks, 2024c).

2.2 Python

Python är ett populärt och användarvänligt programmeringsspråk som är ett tolkat och objektorienterat programmeringsspråk på hög nivå med dynamisk semantik. Python släpptes år 1991 och målet var att göra Python till ett lättförståeligt men effektivt språk, anpassad för vardagliga uppgifter och öppen källkod så att andra kan hjälpa till med utvecklingen. Python används huvudsakligen för att skapa webbsidor, för automation av uppgifter, dataanalys, visualisering av data och tillämpningar inom AI. (Van Deusen, 2023).

Python är ett av flera program som erbjuder ökad produktivitet eftersom det inte kräver något kompileringssteg, har enkla syntax och en inbyggda felsökare, som kan inspektera både lokala och globala variabler på källnivå samt hantera brytpunkter. (Python, 2024).

2.3 Structured Query Language

Structured Query Language eller SQL uppfanns på 1970-talet och bygger på den relationella datamodellen. SQL var också tidigare känt som Relational Software vilket blev den första leverantören som erbjöd ett system för att hantera relationsdatabaser.

SQL är ett programmeringsspråk som är användarvänligt och går lätt att integrera med andra programmeringsspråk, i detta fall Python. SQL används för bearbetning och lagring av data i en relationsdatabas. En relationsdatabas lagrar all sin data i tabellform, med rader och kolumner representerar olika dataattribut samt relationerna mellan datavärdena. SQL-satser kan används för att söka, lagra, uppdatera och radera information från databasen. (Amazon Web Services, 2024b).

För att hantera denna data så använder man sig av SQL-satser, där man kan skriva in olika kommandon för att behandla data. Några vanliga kommandon är till exempel; *SELECT*, *FROM* och *WHERE*. *SELECT* används för att välja vilka kolumner man är intresserade av i tabellen, *FROM* för att välja från vilken datatabell man vill hantera data och *WHERE* för att göra hanteringen mera filtrerad, det plockar upp var ett värde uppfyller ett visst krav och visar inga andra rader var värdet inte förekommer. (Carnes, 2020).

2.4 Apache Spark

Apache Spark är en öppen källkods-analysmotor som stöder flera språk och är optimerad för big data och datavetenskap, inklusive maskininlärning. Med Spark kan du utföra realtidsanalyser och behandla data i omgångar. Spark har fyra inbyggda programmeringsspråk, Python, Scala, Java och R. Spark har dessutom flera olika bibliotek som stöd för att bygga olika applikationer för grafbearbetning med GraphX,

maskininlärning med MLib, SQL samt strömbearbetning. Spark har ett effektivt sätt att hantera data, över ett serverkluster så aggregeras data samt partitioner, där det sedan kan flyttas, beräknas eller köras genom en analysmodell. Spark exekveras väldigt snabbt genom att lagra data i minne över flera samtidiga operationer, genom att minska på antalet läs- och skrivoperationer så ökas bearbetningshastigheten med minnesmotorn. (Databricks, 2024f).

2.4.1 PySpark

PySpark är Python's API för Apache Spark, vilket möjliggör användning av Sparks funktioner i ett Python program. Dataramar kan byggas med hjälp av PySparkSQL, vilket är ett PySpark bibliotek för att kunna göra analyser av stora mängder data med hjälp av SQL-tekniker och SQL-satser. (Databricks, 2024d).

Ett liknande känt datahanteringsbibliotek är Pandas. Spark och Pandas fungerar lite olika när det gäller datahantering, Spark är alltid snabbare än Pandas då det gäller big data medan Pandas kan vara enklare att använda om man har en mindre datamängd (Raj, 2023).

3 Utförande

Detta kapitel handlar om utförandet av examensarbetet samt förklaringar över olika tekniker inom programmering som använts.

3.1 Förberedande arbete

Jag började med att utöka mina kunskaper om Databricks notebooks och instrumentpaneler, vilket innefattar SQL och Python. Den tabell som har använts i detta arbete heter "datatabell" och den innehåller nästan all sensordata från produktionen. Det egentliga namnet på datatabellen har lämnats bort.

Column	Type
machineid	string
sensorid	string
value	string
readtimestamp	timestamp
sourcetimestamp	timestamp

Figur 1. Figur över datatabell.

Som figur 1 visar så finns det fem olika kolumner i tabellen. Två av dessa är tidsstämpelkolumner i tidsstämpelformat: en för när sensorn hämtade data "sourcetimestamp" och den andra för när sensordata kom till databasen "readtimestamp". De återstående tre kolumnerna är av typen sträng och innehåller sensorvärdet "value", maskin-ID "machineid" och sensor-ID "sensorid". Värdet är i strängformat eftersom de flesta sensorer returnerar booleska värden eller siffror.

Det experimenterades först med att göra flera olika notebooks med data från sensorerna, för att få en uppfattning om datamaterialet och för att bli mera bekant med verktyget. Intervjuerna inleddes senare, med hänsyn till de nya kunskaperna.

3.2 Intervjuer

För att först ta reda på vad operatörerna samt analytikerna faktiskt var intresserade av så utfördes fyra intervjuer. Det försöktes ställas likadana frågor till alla. Genom frågorna fick jag reda på vilken data är de mest intresserade av, vilken maskin de är mest intresserad av och annat för att förstå behoven mera på detaljnivå.

3.2.1 Frågor till respondenterna

Vad jobbar du med och hur länge har du jobbat hos Mirka?

Hur bekant är du med Databricks?

Vilken data är du mest intresserad av?

Vilka maskiner är du främst intresserad av?

Vad har du gjort förut för att få denna data?

Vad har ni för behov av verktyget?

3.2.2 Intervju A

Respondent A har arbetat 16 år hos Mirka inom produktkvalitet, livscykelhantering av produkter, varit tidigare laborant och är en novis med Databricks. Hen var mest intresserad av att länka ihop testdata med processdata, till exempel när olika processvärden har varierat så vill hen se vad det påverkar i produktionen och dylikt. Förut gick respondent A till maskinen och skrev ner luftflödet på papper, vilket sedan fördes till en dator och det blev inskrivet i Excel manuellt. Hen ser verktyget som ett stöd för att lära sig mera om processen och vad som påverkar vad och ser det som en fördel att göra allt via ett och samma program.

3.2.3 Intervju B

Respondent B har arbetat som dataanalytiker i 16 år hos Mirka och har lärt sig lite om Databricks under några månader. Datamässigt var hen mest intresserad av processdata korrelerat med utdata, även ES-data, vilket kommer från ett annat företag. Tidigare har respondenten hämtat data genom att ha tagit en USB-sticka till maskinen hen ville ha data

från och sedan fört över informationen till ett program. Vartefter har det kommit flera program som hen har använt för att analysera data.

3.2.4 Intervju C

Respondent C har varit anställd hos Mirka i 29 år och har deltagit i ett flertal projekt med olika befattningar. Även om respondenten är bekant med Databricks så har det ännu inte använts i deras arbete. Med intresse för datamässiga aspekter som produktivitet, kapacitet och gångtidsmätning inom produktionslinjerna, var respondenten intresserad av att optimera effektiviteten. Det inkluderar att ersätta de veckovisa mötena med personalen i produktionen med en live visualisering vid linjen för att tydligt illustrera deras prestation och identifiera faktorer som påverkar gångtiden. Respondent C tog även upp ett exempel där höga övertidskostnader var ett problem.

3.2.5 Intervju D

Respondent D har varit anställd hos Mirka i 11 år och har haft en många roller inom företaget. Hen har arbetat i produktionen, varit arbetsledare och är nu platschef. Respondent D hade ingen tidigare kännedom om Databricks, så en kort introduktion till programmet behövdes. Efter att ha fått insikt i dess möjligheter, insåg hen att kilowattimmar, alltså energiförbrukning per maskin var av stort intresse för både sig själv och den övriga personalen. Detta innebär att det är av betydelse att kunna identifiera vilka maskiner som drar mest ström. I dagsläget erhöll respondenten data i en samlad form varje månad och ett verktyg som kunde erbjuda mer detaljerad och realtidsinformation skulle ha varit av stor nytta.

3.2.6 Summering av intervjuer

För respondenterna skapas två olika verktyg. För respondent A skapades en notebook som hen kan använda för att enkelt få tag på processdata i form av olika typer av sensordata från fabriken. Respondent B kunde eventuellt också ha nytta av notebooken som gjordes på basis av respondent A:s behov. För respondent D gjordes en instrumentpanel. Respondent D var intresserad av energiförbrukningen för olika maskiner i fabriken. För att tillgodose detta behov skapades en instrumentpanel där man enkelt kan visualisera detta. Resten av respondenternas önskemål blir eventuellt förverkligade i framtiden i form av

verktyg inom Databricks eftersom det inte i dagens läge var möjligt att uppfylla deras önskemål.

3.3 Instrumentpanel över energiförbrukning

Som första verktyg gjordes en instrumentpanel över energiförbrukning.

3.3.1 Plan för verktyget

Respondent D var intresserad av energiförbrukning, som kan snabbt visualiseras och där personalen kan se förbrukningsvärdet för olika maskiner i fabriken. Detta behövs för att få en klar uppfattning av hur mycket energi maskinerna förbrukar och när. Genom detta kunde personalen få information om vad kan de ändra på för att effektivisera bland annat maskinernas konsumtion.

3.3.2 Datauppsättningar och SQL-satser

En instrumentpanels data kommer från datauppsättningar. En datauppsättning är en SQL-sats eller direkt ett val av en tabell som användaren måste skapa eller välja (Microsoft, 2024c). Satsen ser ut på följande sätt.

Kodexempel 1. SQL-sats för energiförbrukning del 1

```
WITH agg_data AS (  
  SELECT machineid, sensorid, AVG(CAST(value AS FLOAT)) as avg_value,  
  date_trunc('hour',sourcetime) as h_Time  
FROM datatabell  
WHERE (  
  (machineid = 'maskin 1' AND sensorid = 'specifik sensor')  
  OR  
  (machineid <> 'maskin 1' AND sensorid LIKE '%[kW]')  
)  
  
AND sourcetime > '2024-01-01'  
GROUP BY  
  machineid, sensorid, h_Time  
)
```

Observera att vissa värden har blivit utbytta i kodexemplen från egentliga värden. Först definieras satsen som "agg_data", vilket står för aggregerat data. Inom den väljer vi de kolumner vi är intresserade av från tabellen. Med hjälp av CAST byter vi sedan på datatypen

för värdet, eftersom den ursprungligen är definierad som en sträng så måste det bytas till ett flytvärde (W3schools, 2024i). Sedan tas ett medeltal av värdet, med hjälp av kommandot *AVG*. Kommandon som *SUM* och *AVG* är metoder i SQL för att ändra på numeriska data till en summering eller ett medeltal (W3schools, 2024c). Med hjälp av byte av datatypen kan visualiseringarna förstå att värdet är numeriskt. Medelvärdet definieras sedan som "avg_value".

Slutligen trunkerar vi datumet med *date_trunc* för varje timme, och definierar det som "h_Time". Kommandot *date_trunc* tar emot två variabler, först intervallet man vill ha tiden i och sedan det som ska filtreras till intervallet (Amazon Web Services, 2024a). Sedan tar vi data från vår datatabell där maskin-ID är lika med maskin ett och där sensorernas ID är lika med en specifik sensor vid maskin ett som behandlar totala kilowatt data hos maskinen. Om maskin-ID inte är lika med maskin ett så tar vi upp alla sensorer på vanligt sätt, var de slutar på "[kW]". Vi kontrollerar också att tiden är högre än första januari 2024, eftersom all data före 2024 var irrelevant. En gruppering görs sedan enligt maskin-ID, sensor-ID och tiden. Kommandot *GROUP BY* används eftersom vi aggregerar värdet då vi vill ha medeltalet, kommandot tar och grupperar alla likadana värden tillsammans, som en summering (W3schools, 2024d).

LIKE fungerar som en kontroll där man kan söka efter om en kolumn uppfyller ett visst mönster på värdet. Man kan plocka upp värden som innehåller ett visst mönster genom att lägga procenttecken före, efter eller runtom värdet. Detta används när vi söker efter sensorer som slutar på "[kW]", då vi vill ha likt de värden för att fånga alla rader där mönstret dyker upp i kolumnen för sensor-ID. (W3schools, 2024f).

Kodexempel 2. SQL-sats för energiförbrukning del 2

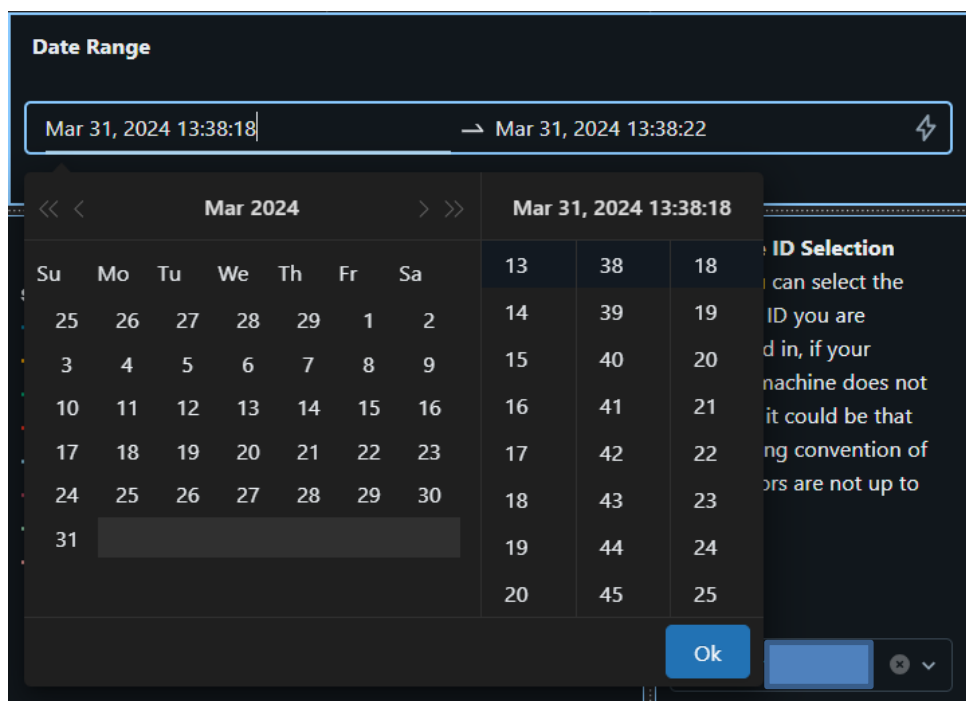
```
SELECT
machineid,
sensorid,
h_Time
avg_value,
CAST((avg_value * pris) AS DECIMAL(10, 1)) as Euro,
CAST(SUM(avg_value) AS DECIMAL(10, 1)) AS kWconsumption

FROM agg_data
GROUP BY machineid, sensorid, avg_value, h_Time
```

Efter de två satserna väljer vi maskin-ID, sensor-ID, aggregerade tiden som vi tidigare definierat som "h_Time" och värdet som vi tidigare definierat som "avg_value". Vi gör två nya kolumner från vårt medelvärde. Den första kolumnen blir priset på energiförbrukningen över tiden med namnet Euro. Det är värdet gånger priset av el, vilket sedan blir bytt till decimalformat. Likadant med kilowattkonsumtion, vilket är summeringen av medelvärdet som decimal. Observera att "pris" egentligen ska vara ett tal, men talet har dolts. Slutligen väljs all data från vår aggregerade datatabell och det görs en gruppering.

3.3.3 Visualisering

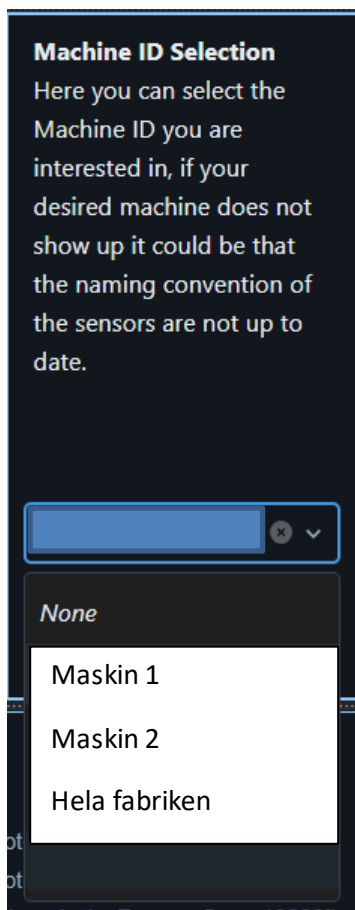
Instrumentpanelen består av sex olika rutor, där en av dem är visualiseringar, två av dem är filtreringsceller, två celler visar numeriska värden och en cell är enbart text. Textcellen beskriver enbart vad verktyget är, sedan följs den av en filtercell där man kan välja sitt datumsintervall.



Figur 2. Filtrering för val av datum.

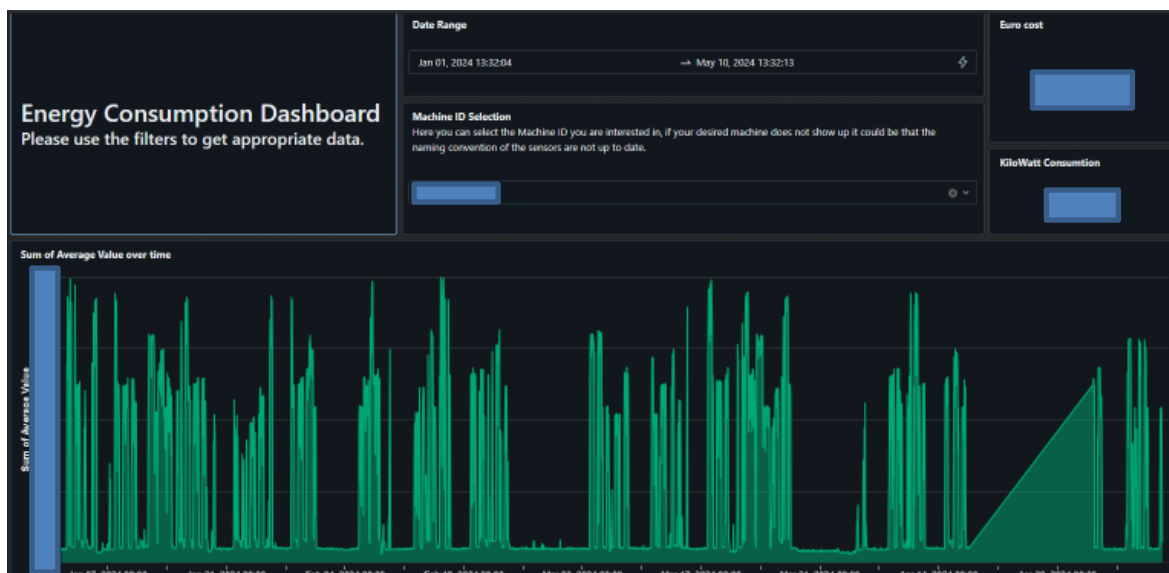
Som det visas i figur 2 så är det enkelt att välja datum och tid. Val av datum och tid lades till så att användaren kan välja från vilken tid de vill se konsumtion. Efter att man har lagt

in sitt datum kan man välja vilken maskin man är intresserad av med hjälp av en till filtreringscell, som är anpassad för maskin-ID.



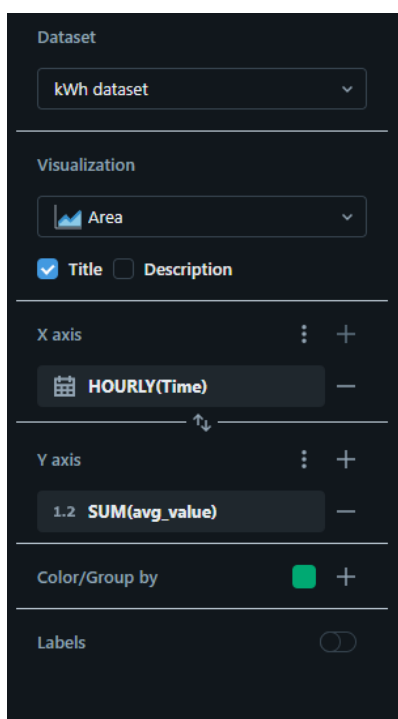
Figur 3. Val av maskin id vid filtrering.

I figur 3 syns endast ett antal olika maskiner för tillfället, vilket har att göra med Mirkas namnkonvention på deras sensorer. Instrumentpanelen är byggd för att fånga upp alla apparater och sensorer som följer den moderna nämningsskonventionen, och med tiden kommer flera maskiner att hittas. Som exempelvärde använder jag "maskin 1", en maskin som har det flesta sensorer uppdaterade till dagens standarder och har en sensor över total kilowatt. Med hjälp av tidsintervallet samt val av maskin-ID uppdateras en visualisering på summan av kilowattimmar.



Figur 4. Instrumentpanel över energiförbrukning från olika maskiner.

Visualiseringen i figur 4 använder sig av medelvärde, gör en summering av det och lägger det på y axeln medan tiden läggs på x-axeln där den visar data inom tidsintervallet användaren gav. Användaren kan lätt zooma in på de områden de är intresserade av. Vi har då även två rutor för kostnaden över tiden samt kilowattkonsumtion över tiden uppe i höger. All data hämtas från samma datauppsättning, i detta fall från SQL-satsen som gjordes.



Figur 5. Konfiguration av visualisering för instrumentpanel.

Figur 5 visar hur visualiseringen är konfigurerad.

3.4 Notebook för maskinsensorer

Denna notebook blev ganska lik en prototyp som jag gjorde tidigare för att bekanta mig med Databricks, vilket ledde till en smidig utveckling av verktyget. Som övning hade jag gjort en notebook som tar upp till två maskin-ID, där man sedan väljer sensor-ID, vilket sedan ger användaren olika visualiseringar efter körningen.

3.4.1 Plan för verktyget

För respondent A valde jag att göra en notebook, var det blir smidigare att hitta fram specifika data jämfört med en instrumentpanel, eftersom det finns många sensorer i fabriken. I detta fall behövdes det två olika SQL-satser. I stället för att en datauppsättning läses in i början måste allt exekveras via kod inom cellerna. Man kan också göra widgetar, som användaren lätt kan använda för att skicka parametrar till koden, vilket sedan förs över till SQL-satsen inuti koden. För att skapa dessa verktyg användes Python, SQL samt Sparks funktioner för att lätt kunna använda SQL i Python.

3.4.2 Databricks-widgetar

I denna notebook lade jag till sex olika widgets, vilket ska underlätta processen genom att användaren smidigt kan välja värden i widgetarna i stället för att ändra värden manuellt i koden.

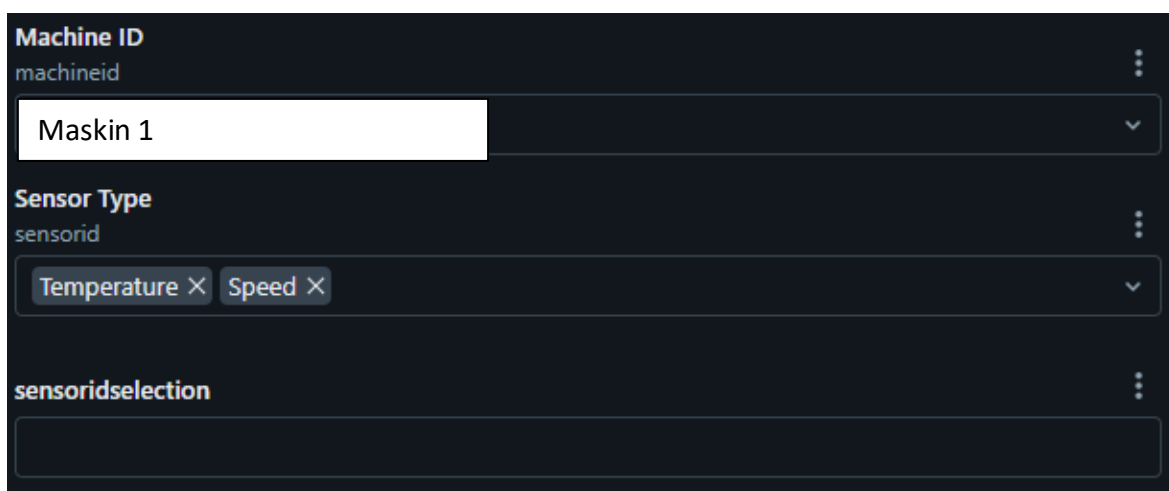
Första widgeten har användaren att välja värdet för maskin-ID, värden är hårdkodade i widgeten eftersom det sällan läggs till en ny maskin i produktionen, detta kan fixas lätt i framtiden genom att enkelt lägga till den nya maskin-ID statistiskt. Om man i stället skulle vilja ha att widgeten bildas i koden med maskin-ID skulle man behöva köra åtminstone en cell för att få detta, vilket gör processen osmidigare.

En till widget gjordes för typ av sensor, vilket har användaren att välja på statiska värden när det gäller vilken typ av data de är intresserad av. Detta blir sedan konverterad i koden till att uppfylla kraven för att SQL-satsen ska få träffar.

Två olika widgetar för tidsstämpel finns också för att få ett visst tidsfönster över den data man är intresserad av. Ifall tidsfönstret är för brett kan det leda till att data blir trunkerat, vilket leder till en approximativ visualisering av data.

Ytterligare en widget gjordes för att kunna välja en specifik sensor. Denna widget måste byggas med hjälp av koden, eftersom vi vill inte statiskt lägga in tusentals sensorer. När den första cellen körs fungerar widgeten efter körningen, eftersom den plockar upp beroende på vilken maskin-ID som har valts så väljer den distinkt alla sensorer vid de maskiner man valt. Sedan då alla sensorer har plockats upp listar den upp alla i en rullgardinsmeny där användaren får välja en specifik sensor.

Slutligen finns det en widget för att välja intervall, det finns att välja på sekunder, minuter, timmar, dagar och rådata. Eftersom data lätt kan bli trunkerat med rader som uppstår nästan varje sekund, så har jag gett användaren möjligheten att få en större överblick över data. Genom att skicka de intervall användaren valt ändras satsen till att den bara plockar upp med de intervall de valt.



The screenshot shows a Jupyter Notebook interface with a dark theme. It contains three vertically stacked widgets for data selection:

- Machine ID**: A dropdown menu with the label 'machineid' and a value of 'Maskin 1'. A three-dot menu icon is on the right.
- Sensor Type**: A multi-select dropdown menu with the label 'sensorid'. It currently shows 'Temperature' and 'Speed' as selected items, each with a close button (X). A three-dot menu icon is on the right.
- sensoridselection**: An empty dropdown menu with a three-dot menu icon on the right.

Figur 6. Widgetar för notebook del 1.

Timestamp Start
timestamp_start
2024-03-28T14:00:00.00

Timestamp End
timestamp_end
2024-03-30T15:00:00.00

Data Interval
input_interval
hour

Figur 7. Widgetar för notebook del 2.

Figur 6 och 7 visar hur alla widgetar ser ut samt med exempelvärden färdigt ifyllda.

3.4.3 Kodan

Notebooken består av två olika celler, ursprungligen bestod den av flera, men efter lite finslipning samt effektivisering av koden behövdes endast två celler. Den första cellen gör största delen av arbetet. Cellen börjar med att ta in alla värden användaren har gett i widgetarna och definierar dem som värden inuti koden.

Kodexempel 3. Inläsning av widgets

```
machineidselect = dbutils.widgets.get("machineid")
machineid = machineidselect.split(",")
sensortypeselect = dbutils.widgets.get("sensorid")
sensortype = sensortypeselect.split(",")
timestamp_start = dbutils.widgets.get("timestamp_start")
timestamp_end = dbutils.widgets.get("timestamp_end")
interval = dbutils.widgets.get("input_interval")
```

Eftersom användaren kan mata in flera värden på två widgetar så måste koden ta emot flera värden, men delar upp dem med kommatecken som separerare. Kodexempel 3 visar hur denna data hämtas in. Först ger vi värden till olika variabler som vi sedan kommer att använda i koden genom att använda oss av Databricks inbyggda funktion *dbutils.widgets.get* för att kalla på de widgets vi gjort för att hämta inmatningarna. Funktionen hämtar exakt vad användaren har matat in, i detta fall kan användaren endast skicka färdiga statistiska värden på alla variabler förutom tidsfiltreringen (Databricks, 2024b). Eftersom användaren väljer flera värden i widgeten för sensortyp och maskin-ID måste vi

delar upp det användaren skickar in, vilket gör den redo för att matas in i satsen. Uppdelningen körs av kommandot *split* och delar upp värden var ett kommatecken uppstår och lägger dem som en lista, en sträng kan bli till ett antal strängar i en lista beroende på hur många ord som finns mellan kommatecknen (Geeksforgeeks, 2024).

Kodexempel 4. Definering av dataram som lista

```
df = []
```

Sedan definieras en tom dataram som en lista, så att vi kan hantera flera dataramar beroende på hur många typer av sensorer användaren har valt.

Beroende på vad användaren valt i widgeten för sensortyp så byts det egentliga värdet så det motsvarar ett värde som passar i SQL-satsen. Satsen kräver att sensortyperna ska sluta på enheternas symbol. Användaren ser då i stället bara simplifierade varianter av sensortypen i widgeten, exempelvis widgeten visar "Temperature" medan satsen kräver "[C]".

Kodexempel 5. Konvertering från användarens input till sensorernas namn

```
for i in range(len(sensortype)):
    if sensortype[i] == "Temperature":
        sensortype[i] = "[C]"
    elif sensortype[i] == "Airflow":
        sensortype[i] = "[m3_h]"
    elif sensortype[i] == "Speed":
        sensortype[i] = "[m_min]"
    elif sensortype[i] == "Humidity":
        sensortype[i] = "[RH%]"
    elif sensortype[i] == "Torque":
        sensortype[i] = "[N]"
```

I koden loopar vi igenom längden av "sensorid" vilket vi hämtade från widgeten. Beroende på längden kontrollerar vi vilket värde som matats in och konverterar det till motsvarande värde. Koden loopar beroende på längden av listan "sensortype". En lista kan ses som en variabel som lagrar många olika värden, i detta fall lagrar vår sensortyp exempelvis; "Temperature", och "Airflow" (W3schools, 2024b). Inuti loopen använder vi oss av *if* och *elif* satser vilket kontrollerar att olika kriterier uppfylls och exekverar kod beroende på vad som blev uppfyllt i loopen (Python, 2024).

Kodexempel 6. Formatering av maskin-ID för SQL-sats

```
ids = ",".join([f"{id}" for id in machineid])
```

De olika maskin-ID som har blivit valda med widgetarna blir då inlästa och konverterade i koden. Vi definierar "ids" som resultatet av en loop som tar och räknar upp alla olika maskin-ID som blivit valda i widgeten och lägger till dem till "ids" med kommatecken som separerare. Detta görs så alla maskinernas ID lagras i en och samma variabel, i detta fall vill vi inte ha det i en lista, som vi gör med sensor-ID. I koden så loopar det igenom alla ID inom maskin-ID. "f" som förekommer före den egentliga loopen konverterar allting efter det till en f-sträng, detta möjliggör tilläggning av data som är lagrad i en variabel om man använder klammerparenteser, i detta fall "{id}" (Jablonski 2023).

Nu fortsätter den centrala delen av koden, vilket exekverar en SQL-sats som ger oss datatabeller och visualiseringar.

Kodexempel 7. SQL-sats för notebook med behandling av tidsintervall

```
if interval != "raw":

    for i in range(len(sensortype)):
        query3 = f"""
            SELECT machineid, sensorid, AVG(CAST(value AS FLOAT)) AS new_value,
            date_trunc('{interval}', sourcetime) AS trunc_date
            FROM datatabell
            WHERE machineid IN ({ids})
            AND sensorid LIKE '%{sensortype[i]}%'
            AND sourcetime BETWEEN '{timestamp_start}' AND '{timestamp_end}'
            GROUP BY machineid, sensorid, trunc_date
            ORDER BY sensorid, trunc_date
            """
        df.append(spark.sql(query3))
        print(sensortype[i])
```

I kodexempel 4 har vi den huvudsakliga satsen, denna förekommer två gånger men i en lite annorlunda form sedan. Vi börjar med att kontrollera om användaren har valt att hen vill ha rådata genom att kontrollera att intervallet inte är lika med "raw". Ifall intervallet inte är "raw" så fortsätter koden med att starta en loop beroende på hur många sensor-ID användaren har matat in via widgeten.

Vi definierar sedan vår SQL-sats vilket vi senare förvandlar till en dataram. I satsen börjar vi med att välja sensor-ID, maskin-ID, ett medeltal av värdet till flytande och sedan "sourcetimestamp" som "trunc_date". På likadant sätt som på instrumentpanelen så blir värdena ändrade till att visa ett medelvärde över intervallet användaren har givit på en rad.

Till näst tar vi från vår datatabell, var maskin-ID är inom "ids", vi söker med */N* var maskin-ID är inom de flera värden som skickats med "ids". Variabeln "ids" innehåller alla maskin-ID som användaren matat in via widgeten (W3schools, 2024e).

Vi tar sedan upp var sensor-ID slutar på vad positionen av sensortypen 'i' innehåller, beroende på hur mycket vi har loopat igenom koden. Sedan kontrollerar vi också att tiden är inom de tidsintervall användaren givit i widgetarna. Vi grupperar sedan enligt maskin-ID, sensor-ID och tiden. Vi sorterar sensor-ID som primär sortering och tiden som sekundär sortering med *ORDER BY*, nu kommer tabellen ha sensorerna i alfabetisk ordning men ändå så att tiden är korrekt med den andra sorteringen (W3schools, 2024g).

Nästa steg av loopen är att göra en dataram genom att använda *spark.sql(query)* inuti *df.append()*. Med *spark.sql* funktionen så förvandlar det man satt inom parenteserna till en dataram, i detta fall lägger vi in "query3", som tidigare i loopen blir definierat som själva SQL-satsen (Apache Spark u.å.). Funktionen *append* lägger till dataramen till vår lista "df", i en position beroende på vilket värde 'i' har (W3schools, 2024a).

Slutligen körs det en print av sensortyperna för att visa användaren exakt vad dom har valt via widgetarna. Detta kan vara till stor nytta om något går snett och det måste felsökas i koden. Direkt efter loopen är färdig så körs också en print på vilka maskin-ID som valts.

Kodexempel 8. SQL-sats för notebook med behandling av rådata

```

elif interval == "raw":
    for i in range(len(sensortype)):
        query = f"""
            SELECT machineid, sensorid, CAST(value as FLOAT) as new_value,
            sourcetime as trunc_date
            FROM datatabell
            WHERE machineid IN ({ids})
            AND sensorid LIKE '%{sensortype[i]}%'
            AND sourcetime BETWEEN '{timestamp_start}' AND '{timestamp_end}'
            ORDER BY sensorid, trunc_date
            """
        df.append(spark.sql(query))
        print(sensortype[i])

```

I kodexempel 8 ser man hur den andra satsen ser ut som behandlar rådata. Den är väldigt lika till den föregående satsen men gör inga aggregeringar och försöker ha samma benämningar som de värden i den föregående satsen. Detta görs eftersom visualiseringarna inte kan anta att ett värde har bytt namn, till exempel skulle den inte förstå att "value" nu heter "new_value" och ska användas i visualiseringarna.

Kodexempel 9. Widget-initialisering från kod

```

query2 = f"SELECT DISTINCT sensorid FROM datatabell WHERE machineid in ({ids})"

```

Efter loopen kört färdigt så kommer lite mera kod för skapning av en widget. Kodexempel 9 visar en SQL-sats som blir definierad med namnet "query2", var vi använder oss av *SELECT DISTINCT*, kommandot *DISTINCT* returnerar endast unika värden på sensor-ID inom vår sökning (W3schools, 2024h). Vi väljer sedan från vår datatabell var maskin-ID är inom "ids".

Kodexempel 10. Definiering av dataram för widget

```

df2 = spark.sql(query2)

```

Vi gör sedan en dataram av SQL-satsen. Detta lagras inte i någon list som de föregående dataramarna gjorde, eftersom vi bara behöver en.

Kodexempel 11. Kod som initialiserar widgeten för specifik sensor

```

dbutils.widgets.combobox("sensoridselection", "", [row[0] for row in df2.collect()])

```

Nu skapas en widget via koden i formen av en kombinationsruta vilket har ID "sensoridselection", vilket man använder sedan för att hämta widgetens värde. Nästa fält är för att lägga till ett standardvärde, eftersom ett standardvärde i detta fall skulle eventuellt leda till förvirring så lämnas fältet tom. Efter det väljs vilka värden användaren ska kunna välja mellan. Vi loopar sedan igenom varje rad i df2, vilket är vår dataram från tidigare som plockar upp distinkta givare hos vald maskin (Databricks 2024a). Vi använder oss av kommandot *collect()* för att hämta data från en dataram så att widgeten ska kunna tolka skilda rader i dataramen (Geeksforgeeks, 2021).

Kodexempel 12. Kod som visar alla visualiseringar

```
for i in range(len(df)):  
    display(df[i])
```

Då all kod i cellen har kört färdigt körs ytterligare en till loop som syns på kodexempel 12, var det loopar *display()* beroende på hur många dataramar det finns. Kommandot *display()* visar tabeller och eventuellt visualiseringar om de finns färdigt konfigurerade som output (Databricks, 2024a). I stället för att ha hårdkodat flera rader eller celler med display hålls programmet mera dynamiskt. Därefter genereras visualiseringarna och tabellerna fram. Tiden för koden att köra kan variera beroende på hur brett data användaren har gett, samt om andra notebooks är i gång på samma datakluster. Om klustret inte är i gång kan det vanligen ta upp till fem minuter att starta det. Det finns många olika faktorer som bidrar till en notebooks prestation.

Då klustret är i gång och man har väntat på att cellen har kört färdigt kan man förvänta sig rimliga resultat beroende på hur noggrann man varit med tidsintervallet och hur många maskiner man valt. Ett problem Databricks har för tillfället är att data blir lätt trunkerat ifall man inte filtrerar sina satser. Max rader man kan ha i en tabell eller visualisering i notebooks är 10 000, men dataramen innehåller fortfarande det egentliga resultatet.



Figur 8. Exempelvisualisering över hastighetssensorer över en viss tidsperiod.

Figur 8 visar en typisk visualisering av flera sensorer, jag valde bara att visa två av de tre som dök upp eftersom de hade varierande data. Det syns nere på figuren att data har blivit trunkerat, detta beror på att det användes rådata som intervall som ett exempel. Denna visualisering kommer att lämna kvar för användarna så att de själv kan analysera den eller välja att lägga till egna vilket också sparas där för andra användare. Visualiseringarna ändrar efter en körning beroende på vad användaren har valt i widgetarna och anpassar sig automatiskt till nytt data.

Kodexempel 13. SQL-sats för visualisering av specifik sensor

```
SELECT sensorid, CAST(value as FLOAT), sourcetime FROM datatabell WHERE
sensorid = '${sensoridselection}' AND sourcetime BETWEEN '${timestamp_start}'
AND '${timestamp_end}'
```

Nästa cell är en snabb SQL-sats, den tar inte i beaktande intervall och visar heller inget maskin-ID. Satsen väljer endast en sensor, vilket användaren måste ange i widgeten högst uppe i notebooken. Satsen tar i beaktande widgets genom att lägga till ett dollartecken före klammerparenteserna genom att hämta widgets (Databricks, 2024b). Denna metod fungerar på samma sätt som då vi i Python cellen använde *dbutils.widgets.get()*. Det lönar sig att använda sig av denna cell när man är ute efter en specifik sensor och vill ha rådata om den.



Figur 9. Exempelvisualisering över specifik hastighetssensor.

Figur 9 visar resultatet av den andra cellen. På alla visualiseringar kan användaren lägga till sina egna visualiseringar och konfigurera dem till deras behov, man kan även zooma in på olika delar av visualiseringen för att få en tydligare blick. Konfigureringsprocessen sköter Databricks smidigt med deras användarvänliga gränssnitt för tabellerna och visualiseringarna. Figur 9 visar då en exempelvisualisering över en givare över en viss tid.

Visualization Editor

Visualization type
Bar

General X axis Y axis Series Colors Data labels

☐ Horizontal chart

X column
Choose column...

Y columns :
Add column

Group by
Choose column...

Error column
Add column

Stacking

☐ Normalize values to percentage

Missing and NULL values
Convert to 0 and display in chart

Figur 10. Visualiseringsredigerare för notebook.

Figur 10 visar hur visualiseringsredigeraren ser ut. Användaren kan utnyttja många olika visualiseringstyper och pröva sig fram tills de får ett önskat resultat. Som en standard implementerades en visualisering för varje sensortyp. Visualiseringarna är helt enkelt medelvärde över tid, med sensorerna som gruppering.

4 Resultat och utvecklingsmöjligheter

Detta kapitel behandlar resultatet av verktygen samt utvecklingsmöjligheterna för dem.

4.1 Resultat

De slutgiltiga verktygen blev en notebook och en instrumentpanel som ännu kan utvecklas till användarnas behov. Med notebooken kan personalen enkelt och effektivt få aktuellt sensordata som de kan analysera. Instrumentpanelen erbjuder en smidig visualisering för analys av energiförbrukning hos maskiner.

Respondent A tyckte att notebooken såg bra ut och hen var nöjd. Däremot saknade hen något data fortfarande från en viss tabell, men detta kommer eventuellt att åtgärdas i framtiden, eftersom min uppgift var bara att behandla sensordata. Respondent A skulle använda verktyget mera då hen har fått tillräckliga rättigheter till notebooken. Däremot har respondent A fått testa på en instrumentpanelsvariant av notebooken över sensordata och var nöjd med resultatet. Notebookens instrumentpanelsvariant skall vidareutvecklas samt den ursprungliga notebooken.

Den simplare SQL-satsen i notebooken som har användaren att använda en specifik sensor tror jag att kommer bli mera använd av de som sköter sensorerna, medan de som kanske inte är intresserade av enskilda sensorer använder först den generella SQL-satsen för att få en helhetsblick var de sedan kan välja vilken sensor de vill ha mera data om.

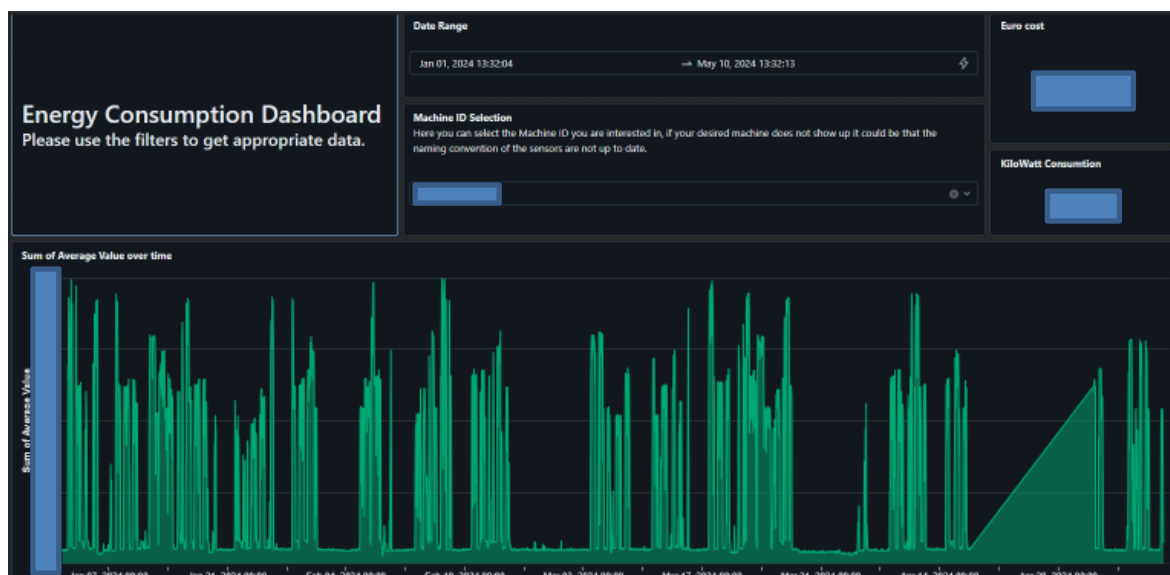
För att göra notebooken så lättanvänd som möjligt har det lagts till text i form av textceller, som försöker förtydliga verktyget för användarna. Det finns även många kommentarsrader inom koden som försöker förklara verktyget stegvis för användaren. Det ska fortfarande läggas till mera kommentarer för användarnas behov, ifall det uppstår funderingar kring notebooken i framtiden från användarna.

Det var utmanande att lista ut hur personalen ville ha instrumentpanelen över energiförbrukning, men vi kom fram till de endast var intresserade av vilka maskiner som konsumerar vad, vilket innebär att ingen specifik sensoranalys är möjlig för instrumentpanelen. Instrumentpanelen är i dagsläget inte redo för feedback.

De slutgiltiga verktygen ser i sin helhet ut på följande sätt:

Kodexempel 14. Fullständig SQL-sats över instrumentpanel

```
WITH agg_data AS (
  SELECT machineid, sensorid, AVG(CAST(value AS FLOAT)) as avg_value,
    date_trunc('hour',sourcetime) as h_Time
  FROM datatabell
  WHERE (
    (machineid = 'maskin 1' AND sensorid = 'specifik sensor')
    OR
    (machineid <> 'maskin 1' AND sensorid LIKE '%[kW]')
  )
  AND sourcetime > '2024-01-01'
  GROUP BY
    machineid, sensorid, h_Time
)
SELECT
  machineid,
  sensorid,
  h_Time,
  avg_value,
  CAST((avg_value * pris) AS DECIMAL(10, 1)) as Euro,
  CAST(SUM(avg_value) AS DECIMAL(10, 1)) AS kWconsumption
FROM agg_data
GROUP BY machineid, sensorid, avg_value, h_Time
```



Figur 11. Slutgiltig bild över instrumentpanel.

The screenshot shows a Databricks notebook interface with several filter widgets at the top:

- Machine ID:** A dropdown menu with 'machineid' selected.
- Sensor Type:** A dropdown menu with 'sensorid' selected, and a tag 'Speed X Temperature X' below it.
- Timestamp Start:** A text input field containing '2024-09-25T14:00:00.00'.
- Timestamp End:** A text input field containing '2024-09-25T15:00:00.00'.
- Data Interval:** A dropdown menu with 'raw' selected.

 Below the widgets is a large documentation cell with the following text:

Enter into the widgets above what you want to filter on.

Please format the timestamp as follows: YYYY-MM-DDTHH:MM:SS.SSS

Example: 2024-01-04T10:25:04.259

Databricks notebooks truncates its rows at 10,000, meaning it cuts out data in the middle of the real table and downsizes it to 10,000.

You can avoid getting truncated values by: Narrowing your time window in timestamp start and end, changing the Data interval or selecting less machines. Keep in mind that selecting less sensortypes ONLY speeds up the whole process and does not give you less data in any way. I recommend to not use the "raw" on data interval if your timestamp is too broad, you can use the cell below the main cell to get raw data of a specific sensor, after running the main cell.

Feel free to add a visualization or data profile of a table and naming it what you want, just make sure someone else has not already made the exact same visualisation.

Running the cells for the first time might take some time but runs after that should be smoother.

Keep in mind that after changing the values in the widgets, even after you've run the code once, you will have to run it again to get appropriate visualisations.

When running the cell below, if it asks for you to start or connect to the cluster, simply select "Start, attach and run"

Figur 12. Bild av notebook widgets samt dokumentationscell.

Kodexempel 15. Fullständig kod från första cellen ur notebooken

```
machineidselect = dbutils.widgets.get("machineid")
machineid = machineidselect.split(",")
sensortypeselect = dbutils.widgets.get("sensorid")
sensortype = sensortypeselect.split(",")
timestamp_start = dbutils.widgets.get("timestamp_start")
timestamp_end = dbutils.widgets.get("timestamp_end")
interval = dbutils.widgets.get("input_interval")

df = []

for i in range(len(sensortype)):
    if sensortype[i] == "Temperature":
        sensortype[i] = "[C]"
    elif sensortype[i] == "Airflow":
        sensortype[i] = "[m3_h]"
    elif sensortype[i] == "Speed":
        sensortype[i] = "[m_min]"
    elif sensortype[i] == "Humidity":
        sensortype[i] = "[RH%]"
    elif sensortype[i] == "Torque":
        sensortype[i] = "[N]"

ids = ",".join([f'"{id}"' for id in machineid])

if interval != "raw":
    for i in range(len(sensortype)):
        query3 = f"""
```

```

        SELECT machineid, sensorid, AVG(CAST(value AS FLOAT)) AS new_value,
date_trunc('{interval}', sourcetime) AS trunc_date
        FROM datatabell
        WHERE machineid IN ({ids})
        AND sensorid LIKE '%{sensortype[i]}'
        AND sourcetime BETWEEN '{timestamp_start}' AND '{timestamp_end}'
        GROUP BY machineid, sensorid, trunc_date
        ORDER BY sensorid, trunc_date
        """

        df.append(spark.sql(query3))
        print(sensortype[i])

elif interval == "raw":
    for i in range(len(sensortype)):
        query = f"""
        SELECT machineid, sensorid, CAST(value as FLOAT) as new_value,
sourcetime as trunc_date
        FROM datatabell
        WHERE machineid IN ({ids})
        AND sensorid LIKE '%{sensortype[i]}'
        AND sourcetime BETWEEN '{timestamp_start}' AND '{timestamp_end}'
        ORDER BY sensorid, trunc_date
        """

        df.append(spark.sql(query))
        print(sensortype[i])

print(ids)

query2 = f"SELECT DISTINCT sensorid FROM datatabell WHERE machineid in ({ids})"

df2 = spark.sql(query2)

dbutils.widgets.combobox("sensoridselection", "", [row[0] for row in df2.collect()])
for i in range(len(df)):
    display(df[i])

```

Kodexempel 16. Fullständig kod från den andra cellen ur notebooken

```

SELECT sensorid, CAST(value as FLOAT), sourcetime FROM datatabell WHERE
sensorid = '${sensoridselection}' AND sourcetime BETWEEN '${timestamp_start}'
AND '${timestamp_end}'

```

4.2 Utvecklingsmöjligheter för verktygen

Notebooken kan utvecklas på en mångfald olika sätt. Koden kan göras effektivare och mer lättförståelig för personalen. Vid widgeten över tidsfiltrering kan det bli oklart hur man skall formatera tiden ifall användaren inte läst noga igenom texten. I framtiden kanske det kan komma en uppdatering för en widget över datum och tid, som kan ge användaren ett smidigare sätt att välja tiden, likadant som på instrumentpanelen. På instrumentpanelerna finns det ett inbyggt filtreringsverktyg för datum och tid som notebooken skulle kunna ha nytta av. Det är möjligt att i framtiden kommer Databricks att implementera detta i sina notebooks. En vidareutveckling kan vara att man gör en instrumentpanelsvariant av notebooken. Den skulle då fungera som en simplificerad variant av notebooken, som inte behandlar lika avancerat data.

Instrumentpanelen skall vidareutvecklas i samband med OT-teamet som sköter sensorerna. När en ny sensor över energiförbrukning lagts till så vidareutvecklas instrumentpanelen.

5 Diskussion

Detta kapitel utgör en allmän diskussion kring hela arbetet.

Då jag påbörjade mitt arbete med projektet ansåg vi att det var mest centralt att jag skapar en plan för hur jag ville arbeta. Jag började först med att lära mig mera om Databricks instrumentpaneler och notebooks. Efter att ha lärt mig en hel del inom en tid bestämde vi datum för intervjuerna och efter dem tog vi ett kort möte om hur vi ska hantera kraven av respondenterna. Några respondenter hade problem som inte gick att lösa i dagens läge. Vi kom då fram till att en del av deras önskemål blir framtida projekt medan resten kan göras nu, åtminstone grunden. Då gjorde jag instrumentpanelen för energiförbrukning samt notebooken för analys av maskinsensorer.

En svårighet med instrumentpanelen var benämningen av sensorerna, det är ganska otydligt om vilken energiförbrukningsgivare som mäter vad. I framtiden hoppas jag att detta blir lättare att tolka men maskin ett hade redan ett bra system för energiförbrukning och i framtiden ska alla maskiner försöka följa upplägget av givarna vid maskin ett då det gäller benämning.

En svårighet med notebooken är att hitta lösningar till olika datatrunkeringsproblem i Databricks. Antalet rader i katalogen för sensorer är väldigt många och det kommer nytt data varje sekund. En annan utmaning har varit att försöka hålla sin kod effektiv. Användare vill inte sitta å vänta cirka fem minuter på att deras kod ska exekvera. Jag hade testat med pandas på en notebook och jämfört hur mycket längre den koden tar att köra och det tog betydligt längre än för spark. Detta blir ett eventuellt uppdrag för framtiden.

Notebooken kommer att vara till nytta för fler än respondenten eftersom jag redan hört på möten av personal som skulle ha behov av ett enkelt verktyg som behandlar sensordata och som är användarvänligt. Deras metod på Databricks har varit tidigare att fråga assistenten att bygga upp en SQL-sats åt dem i ett satsverktyg. Detta innebar att de behövde ta bort eller editera sin sats beroende på vad de ville analysera.

Jag har lärt mig mycket under arbetets gång. Jag har fått mycket kunskap om verktygen, vad de har för betydelse och att man alltid kan göra sina verktyg effektivare. Jag har även fått insikter i hur man intervjuar människor för att få tydliga svar och hur man håller

samtalet på rätt spår. Jag har lärt mig om hur viktigt det kan vara att arbeta utgående från en plan och att komma ihåg att man alltid kan fråga efter hjälp av sina kollegor.

Jag kommer att fortsätta arbeta hos Mirka efter examensarbetet och utvidga mina kunskaper inom området samt göra mera verktyg och vidareutveckla de existerande verktygen. Detta arbete har gett mig självförtroende inför kommande projekt och jag har redan fått liknande uppdrag hos företaget.

6 Källförteckning

Amazon Web Services. (2024a). *DATE_TRUNC function*. Amazon

https://docs.aws.amazon.com/redshift/latest/dg/r_DATE_TRUNC.html

Amazon Web Services. (2024b). *What is SQL (Structured Query Language)?* Amazon

<https://aws.amazon.com/what-is/sql/>

Apache Spark. (u.å.). *pyspark.sql.Session.sql*. Apache Spark.

<https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Session.sql.html>

Carnes, B. (2020, 1 januari). *Basic SQL Commands – The List of Database Queries and Statements You Should Know*. freeCodeCamp.

<https://www.freecodecamp.org/news/basic-sql-commands/>

Databricks. (2024a). *Apache Spark™ Tutorial: Getting Started with Apache Spark on*

Databricks. Databricks. <https://www.databricks.com/spark/getting-started-with-apache-spark/dataframes>

Databricks. (2024b). *Databricks widgets*. Databricks.

<https://docs.databricks.com/en/notebooks/widgets.html>

Databricks. (2024c). *Jupyter Notebook*. Databricks.

<https://www.databricks.com/glossary/jupyter-notebook>

Databricks. (2024d). *PySpark*. Databricks. <https://www.databricks.com/glossary/pyspark>

Databricks. (2024e). *What is a data lakehouse?* Databricks.

<https://docs.databricks.com/en/lakehouse/index.html>

Databricks. (2024f). *What Is Apache Spark?* Databricks.

<https://www.databricks.com/glossary/what-is-apache-spark/>

Geeksforgeeks. (2021). *PySpark Collect() – Retrieve data from DataFrame*.

<https://www.geeksforgeeks.org/pyspark-collect-retrieve-data-from-dataframe/>

Geeksforgeeks. (2024). *Python String split()*. <https://www.geeksforgeeks.org/python-string-split/>

Jablonski, J. (2024, 18 oktober). *Python's F-String for String Interpolation and Formatting*. Real Python. <https://realpython.com/python-f-strings/>

Microsoft. (2024a). *Databricks SQL dashboards*. Microsoft. <https://learn.microsoft.com/en-us/azure/databricks/sql/user/dashboards/>

Microsoft. (2024b). *Introduction to Databricks notebooks*. Microsoft. <https://learn.microsoft.com/en-us/azure/databricks/notebooks/>

Microsoft. (2024c). *What is Azure Databricks?* Microsoft. <https://learn.microsoft.com/en-us/azure/databricks/introduction/>

Mirka. (2024). *Om Mirka*. Mirka. <https://www.mirka.com/sv/fi/top/About-us/>

OpenAI. (2024). *ChatGPT (GPT-3.5) [Large language model]*. <https://chat.openai.com/chat>

Python (2024). *More Control Flows*. Python Software Foundation. <https://docs.python.org/3/tutorial/controlflow.html>

Raj, A. (2023, 30 mars). *PySpark vs Pandas: Performance, Memory Consumption and Use Cases*. Codeconquest. <https://www.codeconquest.com/blog/pyspark-vs-pandas-performance-memory-consumption-and-use-cases/>

Van Deusen, A. (2023, 31 januari) *Python Popularity: The Rise of A Global Programming Language*. Flatiron School. <https://flatironschool.com/blog/python-popularity-the-rise-of-a-global-programming-language/>

W3schools. (2024a). *Python List append() Method*. W3schools. https://www.w3schools.com/python/ref_list_append.asp

W3schools. (2024b). *Python Loop Lists*. W3schools. https://www.w3schools.com/python/python_lists_loop.asp

W3schools. (2024c). *SQL Count(), AVG() and SUM() Functions*. W3schools.

https://www.w3schools.com/sql/sql_count_avg_sum.asp

W3schools. (2024d). *SQL GROUP BY STATEMENT*. W3schools.

https://www.w3schools.com/sql/sql_groupby.asp

W3schools. (2024e). *SQL IN Operator*. W3schools.

https://www.w3schools.com/sql/sql_in.asp

W3schools. (2024f). *SQL LIKE Operator*. W3schools.

https://www.w3schools.com/sql/sql_like.asp

W3schools. (2024g). *SQL ORDER BY Keyword*. W3schools.

https://www.w3schools.com/sql/sql_orderby.asp

W3schools. (2024h). *SQL SELECT DISTINCT Statement*. W3schools.

https://www.w3schools.com/sql/sql_distinct.asp

W3schools. (2024i). *SQL Server CAST() Function*. W3schools.

https://www.w3schools.com/sql/func_sqlserver_cast.asp