POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

EICTA - 2nd Project Delivery - Neo4j and BPMN

Group 10:

Luigi Tiberio [redacted]
Stefano Moreschi
Mattia Silvestri

Academic Year: 2025/2026

Contents

1 | Neo4j

1.1. Database Schema

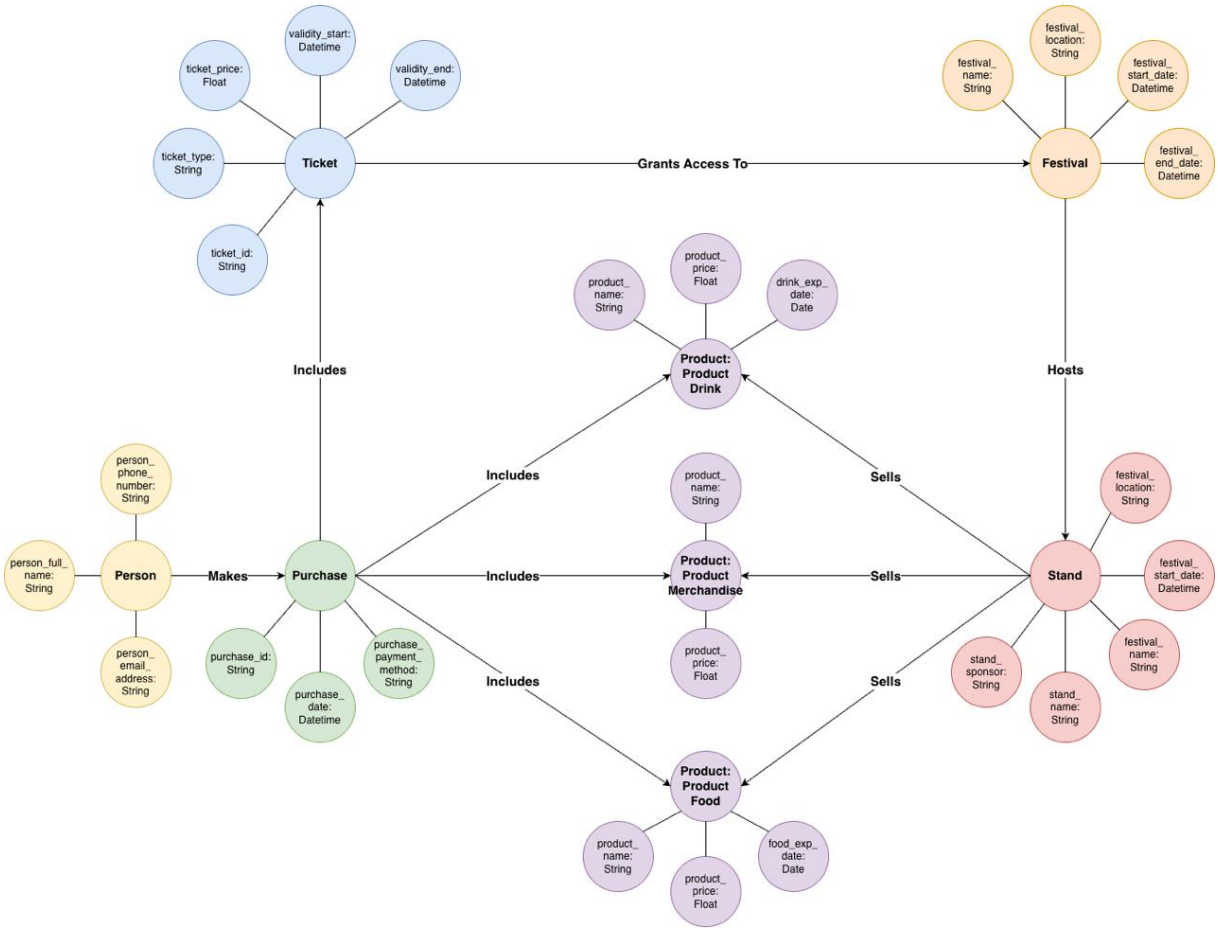

Figure 1. Neo4j diagram

1.2. Database Schema details and considerations

We have decided to create the database following a specific logic: we used the Entities that may represent all the purchase processes that a person is able to do at a festival organized by Euphoric Events. Thus, a sort of "sales tracking" mock database is generated. The Entities and relative attributes considered are the following (entities are **bolded**, attributes are in *cursive*):

- **Person**: "To buy tickets, a person must state their full name, e-mail address, and phone number."
- **Purchase**: "Whenever a purchase happens, the system should store the *date* and the *payment method* used."
- **Ticket**: "To enter a festival, people must buy tickets, which have a *type* (e.g., regular, day pass, VIP), a *price*, and a *validity period*."
- **Festival**: "A festival is characterized by a *name*, a *location*, a *start date*, and an *end date*."
- **Product Drink, Product Food and Product Merchandise**: "A product is described by a *name* and a *price*; furthermore, food and drinks have an *expiration date*, and only food items have a *type* (e.g., vegetarian)."
- **Stand**: "Each stand has a *name*, a *sponsor*.

Along with the Entities, the corresponding relationships were imported. The flow of the relationship follows the logic indicated in the original text and in our previous work:

- **Makes**: "A person can buy..."; *flow*: Person -> Purchase.
- **Includes**: "...multiple tickets, and they can also purchase shop items during the festival."; *flow*: Purchase -> Ticket/Product Drink/Product Food/Product Merchandise.
- **Grants Access To**: "To enter a festival, people must buy tickets."; *flow*: Ticket -> Festival.
- **Hosts**: "Festivals can host several stands."; *flow*: Festival -> Stand.
- **Sells**: "Each stand can sell either of the three aforementioned products."; *flow*: Stand -> Product Drink/Product Food/Product Merchandise.

Finally, some adjustments and considerations had to be made in order to deliver a coherent and polished model:

- **Hierarchy transfiguration**: the original ISA hierarchy of Product is represented by transforming the three Product subtypes (Drink, Food, Merchandise) in three node labels with their respective attributes, but maintaining the original relationships. Now, 'Includes' and 'Sells' are split, respectively, in three different branches, each with the same name: the nature and subject of the relationship is the same, but the object is different. Intuitively, "Includes-Product Drink" is a different relationship than "Includes-Product Food".
- **IDs handling**: Neo4j automatically assigns an internal identifier, unique across the entire database, to each node. Keeping the originally created IDs causes some nodes to have two unique identifiers. Additionally, purchase_id and ticket_id are not directly retrieved from the text, but are gimmick keys generated by us in our previous work to ensure transactions and ticket uniqueness. To guarantee coherence with the original ER model architecture and to avoid data mismatch during data importation, we have decided to keep the IDs. These IDs nevertheless assure coherence with the first work and grant minimal structural changes of the model.

1.3. Neo4j Queries

This section presents all the required queries, each introduced with its specific title, code, explanation, and corresponding output. The outputs were obtained by executing the queries on the provided database dump (*Official Project2.dump*).

1.3.1. Database Modification Queries

The following queries (CREATE, UPDATE, DELETE) modify definitely our Neo4j database by, respectively, creting, updating or deleting existing nodes, relationships or attributes. We have run these queries in a different session than the the one of the Data Retrieval Queries in order not to compromise the original data integrity.

CREATE a "makes" relationship between the User "mdownage0" and the Purchase "P-384107":

```
MATCH (p:person), (pu:purchase)
WHERE p.person_email_address = 'mdownage0@usa.gov'
AND pu.purchase_id = 'P-384107'
CREATE (p)-[r:makes]->(pu)
RETURN r;
```

Output:


```
{
  "identity": {
    "low": 1987,
    "high": 268436480
  },
  "start": {
    "low": 1987,
    "high": 0
  },
  "end": {
    "low": 998,
    "high": 0
  },
  "type": "makes",
  "properties": {},
```

```
        "elementId": "5:59effdac-fdee-44f2-84c5-
1c6bbbf717ed:1152925902653360067",
        "startNodeElementId": "4:59effdac-fdee-44f2-84c5-
1c6bbbf717ed:1987",
        "endNodeElementId": "4:59effdac-fdee-44f2-84c5-
1c6bbbf717ed:998"
      }
```


Figure 1.1: Query 1 CREATE output

CREATE a "involves" relationship between the Purchase "P-279534" and the "Keychain" bought:

MATCH (pu:purchase), (pr:product)

WHERE pu.purchase_id = 'P-279534'

AND pr.product_name = 'Keychain'

MERGE (pu)-[r:includes]->(pr)

RETURN r;

Output:


```
{
  "identity": {
    "low": 1009,
    "high": 268960512
  },
  "start": {
    "low": 1009,
    "high": 0
  },
  "end": {
    "low": 383,
    "high": 0
  },
  "type": "includes",
  "properties": {},
```

```
        "elementId": "5:59effdac-fdee-44f2-84c5-
1c6bbbf717ed:1155176602955416561",
        "startNodeElementId": "4:59effdac-fdee-44f2-84c5-
1c6bbbf717ed:1009",
        "endNodeElementId": "4:59effdac-fdee-44f2-84c5-
1c6bbbf717ed:383"
      }
```


Figure 1.2: Query 2 CREATE output

UPDATE the price to 25 of the Product "Baseball Cap" :

```
MATCH (p:product)
WHERE p.product_name = 'Baseball Cap'
SET p.product_price = 25.0
RETURN p;
```

Output:


```
{
  "identity": {
    "low": 399,
    "high": 0
  },
  "labels": [
    "product",
    "merchandise"
  ],
  "properties": {
    "product_price": 25,
    "product_name": "Baseball Cap"
  },
  "elementId": "4:59effdac-fdee-44f2-84c5-
1c6bbbf717ed:399"
}
```


Figure 1.3: Query 3 UPDATE output

UPDATE to VIP the Ticket TCK1280:

```
MATCH (t:ticket)
WHERE t.ticket_id = 'TCK1280'
SET t.ticket_type = 'VIP'
RETURN t;
```

Output:

```
{
  "identity": {
    "low": 543,
    "high": 0
  },
  "labels": [
    "ticket"
  ],
  "properties": {
    "ticket_type": "VIP",
    "ticket_validity_start": {
      "year": {
        "low": 2024,
        "high": 0
      },
      "month": {
        "low": 1,
        "high": 0
      },
      "day": {
        "low": 17,
        "high": 0
      }
    },
    "ticket_id": "TCK1280"
  },
  "elementId": "4:59effdac-fdee-44f2-84c5-1c6bbbf717ed:543"
}
```


Figure 1.4: Query 4 UPDATE output

DELETE the Purchase "P-932104" of the Ticket "TCK1723":

MATCH (pu:purchase)-[r:includes]->(t:ticket)

WHERE pu.purchase_id = 'P-932104'

AND t.ticket_id = 'TCK1723'

DELETE r;

Output: *"Deleted 1 relationship"*

1.3.2. Data Retrieval Queries

1.3.2.1. MATCH statements and conditions

Return Stand who sells socks:

```
MATCH (s:stand)-[:sells]->(p:product)
WHERE p.product_name = 'Socks'
RETURN s.stand_name, s.festival_name, s.festival_location, s.festival_start_date
ORDER BY s.stand_name;
```

Query Description: The query identifies all stands that sell the product "Socks", returning for each the stand name, the related festival, its location, and start date. It searches for `:stand` nodes connected through the `:sells` relationship to `:product` nodes with `product_name = 'Socks'`, and orders the results alphabetically by stand name.

Query Result:

s.stand_name	s.festival_name	s.festival_location	s.festival_start_date
Amplify Eats	Electric Valley	Barcellona	2024-11-12
BeatStreet Stand	Midnight Grove	Berlin	2024-02-23
Encore Eats	Northern Lights	Monaco	2025-08-13
Groove & Brew	Velvet Noise	Tokyo	2025-09-04
Harmony Hut	Sunset Echo	Tokyo	2024-04-21
Rock 'n' Roll Refresh	Desert Mirage	Milan	2024-05-15
Sonic Snack	Electric Valley	Monaco	2024-01-16

Figure 1.5: Query 1 output

Tickets of the Festival "Northern Lights - Monaco - 13/08/2025":

```
MATCH (t:ticket)-[:grants_access_to]->(f:festival)
WHERE f.festival_name = 'Northern Lights'
AND f.festival_location = 'Monaco'
AND f.festival_start_date = '2025-08-13'
RETURN t.ticket_id, t.ticket_type, t.ticket_price
ORDER BY t.ticket_id;
```

Query Description: The query retrieves all tickets associated with the festival "Northern Lights" held in Monacostarting on 2025-08-13. It searches for `:ticket` nodes connected to the corresponding `:festival` node through the `:grants_access_to` relationship, filters the results by the specified festival attributes, and returns each ticket's ID, type, and price, ordered by ticket ID.

Query Result:

t.ticket_id	t.ticket_type	t.ticket_price
TCK1260	DAY-PASS	10.0
TCK1268	DAY-PASS	10.0
TCK1271	DAY-PASS	10.0
TCK1297	DAY-PASS	10.0
TCK1305	DAY-PASS	10.0
TCK1318	DAY-PASS	10.0
TCK1335	DAY-PASS	10.0
TCK1350	DAY-PASS	10.0
TCK1391	DAY-PASS	10.0
TCK1435	REGULAR	20.0
TCK1447	REGULAR	20.0
TCK1452	REGULAR	20.0
TCK1485	REGULAR	20.0
TCK1491	REGULAR	20.0
TCK1508	REGULAR	20.0
TCK1539	REGULAR	20.0
TCK1557	REGULAR	20.0
TCK1599	REGULAR	20.0
TCK1600	REGULAR	20.0
TCK1633	VIP	50.0
TCK1640	VIP	50.0
TCK1649	VIP	50.0
TCK1673	VIP	50.0
TCK1683	VIP	50.0
TCK1687	VIP	50.0
TCK1697	VIP	50.0
TCK1702	VIP	50.0
TCK1716	VIP	50.0

Figure 1.6: Query 2 output

1.3.2.2. MATCH without a WITH statement

Number of Products sold per Stand at the Festival "Neon Pulse - Monaco - 27/12/2024":

```
MATCH (f:festival)-[:hosts]->(s:stand)-[:sells]->(p:product)
WHERE f.festival_name = 'Neon Pulse'
AND f.festival_location = 'Monaco'
AND f.festival_start_date = '2024-12-27'
RETURN s.stand_name, count(p) AS products_sold;
```

Query Description: The query calculates the number of products sold by each stand participating in the "Neon Pulse" festival held in Monaco starting on 2024-12-27. It matches `:festival` nodes connected via `:hosts` to `:stand` nodes, which in turn are linked through `:sells` relationships to `:product` nodes. The query counts the number of products associated with each stand and returns the stand name along with the total count of products sold.

Query Results:

s.stand_name	products_sold
Harmony Hut	5
Sonic Snack	9
Echo Merch & Munch	4

Figure 1.7: Query 3 output

Stand revenue of the Festival "Neon Pulse - Monaco - 27/12/2024":

```
MATCH (f:festival)-[:hosts]->(s:stand)-[:sells]->(p:product)
WHERE f.festival_name = 'Neon Pulse'
AND f.festival_location = 'Monaco'
AND f.festival_start_date = '2024-12-27'
RETURN s.stand_name, sum(p.product_price) AS total_revenue
ORDER BY total_revenue DESC;
```

Query Description:

The query calculates the total revenue generated by each stand at the "Neon Pulse" festival held in Monaco starting on 2024-12-27. It matches `:festival` nodes linked to `:stand` nodes through the `:hosts` relationship, and further to `:product` nodes via the `:sells` relationship. The query sums the `product_price` values of all products sold by each stand to compute their total revenue, returning the stand name and the corresponding total, ordered in descending order of revenue.

Query Result:

s.stand_name	total_revenue
Sonic Snack	93.4
Harmony Hut	28.1
Echo Merch & Munch	20.5

Figure 1.8: Query 4 output

1.3.2.3. MATCH with one WITH statement

Ticket sales analysis of the Festival "Midninght Grove - Berlin - 23/02/2023:

```
MATCH (t:ticket)-[:grants_access_to]->(f:festival)
WHERE f.festival_name = 'Midnight Grove'
AND f.festival_location = 'Berlin'
AND f.festival_start_date = '2024-02-23'
WITH t.ticket_type AS type, avg(t.ticket_price) AS avg_price, count(*) AS tickets_sold
RETURN type, tickets_sold, avg_price
ORDER BY avg_price DESC;
```

Query Description:

The query analyzes ticket sales for the "Midnight Grove" festival held in Berlin starting on 2024-02-23. It matches `:ticket` nodes connected to the corresponding `:festival` node

through the `:grants_access_to` relationship and groups the results by `ticket_type`. For each ticket type, it calculates the average ticket price and the number of tickets sold, returning these values ordered by average price in descending order.

Query Result:

type	tickets_sold	avg_price
VIP	5	50.0
REGULAR	9	20.0
DAY-PASS	8	10.0

Figure 1.9: Query 5 output

Summary of the Ticket Purchases of the User "zdugget1s":

```
MATCH (p:person {person_email_address: toLower('zdugget1s@unicef.org')})
-[:makes]->(pu:purchase)-[:includes]->(t:ticket)
WITH pu, count(t) AS tickets_count, sum(t.ticket_price) AS purchase_total
RETURN pu.purchase_id, pu.purchase_date, tickets_count, purchase_total
ORDER BY pu.purchase_date;
```

Query Description:

The query provides a summary of ticket purchases made by the person with the email address zdugget1s@unicef.org. It matches the `:person` node (identified by their lowercase email) connected through the `:makes` relationship to `:purchase` nodes, which are in turn linked via `:includes` relationships to `:ticket` nodes. For each purchase, the query counts the number of tickets bought and sums their prices to calculate the total expenditure, returning the purchase ID, date, number of tickets, and total amount spent, ordered by purchase date.

Query Result:

pu.purchase_id	pu.purchase_date	tickets_count	purchase_total
P-903527	2024-08-09	1	10.0
P-694832	2025-07-11	1	50.0
P-976205	2025-11-21	1	50.0

Figure 1.10: Query 6 output

1.3.2.4. MATCH with at least 3 Nodes and multiple WITH statements

Total Revenue and Products Sold per Stand in the Festival "Electric Valley - Madrid - 01/07/2024":

```
MATCH (f:festival {
festival_name: 'Electric Valley',
festival_location: 'Madrid',
festival_start_date: '2024-07-01'
})
MATCH (f)-[:hosts]->(s:stand)-[:sells]->(p:product)
MATCH (pu:purchase)-[:includes]->(p)
WITH s, p // 1° WITH
WITH s, count(p) AS products_sold, // 2° WITH
sum(p.product_price) AS total_revenue
RETURN s.stand_name, total_revenue, products_sold
ORDER BY total_revenue DESC;
```

Query Description:

The query calculates, for each stand participating in the "Electric Valley" festival held in Madrid starting on 2024-07-01, both the total revenue and the number of products sold. It matches the specified `:festival` node connected via the `:hosts` relationship to `:stand` nodes, which in turn are linked through `:sells` to `:product` nodes. Each product is further connected to purchases through the `:includes` relationship. The query then counts the number of products sold and sums their prices to compute the overall revenue per stand, returning these values ordered by total revenue in descending order.

Query Result:

s.stand_name	total_revenue	products_sold
Harmony Hut	471.0	60
Bassline Bites	257.0	65

Figure 1.11: Query 7 output

People who attended at least 3 Festivals since the start of 2024:

```
MATCH (pers:person)-[:makes]->(pu:purchase)-[:includes]->(t:ticket)-[:grants_access_to]
->(f:festival)
WHERE f.festival_start_date >= '2024-01-01'
WITH pers, f, min(t.ticket_validity_start) AS first_use, max(t.ticket_validity_end)
AS last_use
WITH pers, count(DISTINCT f) AS festivals_attended,
min(first_use) AS first_ticket_valid_from,
max(last_use) AS last_ticket_valid_to
WHERE festivals_attended >= 3
RETURN pers.person_email_address AS person, festivals_attended,
first_ticket_valid_from, last_ticket_valid_to
ORDER BY festivals_attended DESC, person;
```

Query Description:

The query identifies all people who have participated in three or more festivals since January 1, 2024. It follows the path from each `:person` through their `:purchase` and `:ticket` nodes to the associated `:festival` nodes via the `:grants_access_to` relationship. For each person, it calculates the number of distinct festivals attended and determines the earliest and latest ticket validity dates. The query then filters for individuals who attended at least three festivals and returns their email address, total number of festivals, and the range of their ticket validity, ordered by attendance and email.

Query Result:

person	festivals_attended	first_ticket_valid_from	last_ticket_valid_to
nhenrichs56@adobe.com	3	2024-01-19	2025-08-22
olindberg2a@cloudflare.com	3	2024-01-20	2025-06-27
pnorledge6t@artisteer.com	3	2024-05-15	2025-06-17
rsperling4f@amazon.co.uk	3	2024-04-21	2024-12-31
tceller3y@t-online.de	3	2024-02-23	2024-11-09

Figure 1.12: Query 8 output

1.3.2.5. MATCH with variable-lenght paths

We explored two different interesting possibilities:

Paths from Person to a "Hot Dog":

MATCH p = (pers:person)-[:makes|includes|grants_access_to|hosts|sells*2..4]->(:product {product_name: 'Hot Dog'})

RETURN pers.person_email_address AS person,

count(p) AS hotdog_paths,

min(length(p)) AS min_hops

ORDER BY hotdog_paths DESC, person;

Query Description:

The query identifies the possible paths connecting each `:person` node to the `:product` node representing *Hot Dog*. It explores relationship chains composed of `:makes`, `:includes`, `:grants_access_to`, `:hosts`, and `:sells`, considering paths 2 to 4 hops long. For each person, it counts how many such paths exist and determines the shortest one, returning the person's email, the total number of "Hot Dog" paths, and the minimum hop distance.

Query Result:

person	hotdog_paths	min_hops
bboss19@reuters.com	2	2
bvowden1@nhs.uk	1	2
ccapey10@marketwatch.com	1	2
ccarsonv@ft.com	1	2
csanzio1c@zimbio.com	1	2
elowisp@sfgate.com	1	2
sgamwell13@tinyurl.com	1	2

Figure 1.13: Query 9 output

Paths from Person to the Festival taken in Naples and started after 28/05/2025:

```
MATCH p = (pers:person)-[:makes|includes|sells|hosts|grants_access_to*3..7]-(f:festival)
WHERE f.festival_location = 'Naples'
AND f.festival_start_date >= '2025-05-28'
RETURN pers.person_email_address AS person,
count(DISTINCT f) AS festivals_reached,
min(length(p)) AS min_hops
ORDER BY festivals_reached DESC, person;
```

Query Description:

This query searches for all paths linking each `:person` node to `:festival` nodes located in *Naples* and starting on or after *2025-05-28*. It follows the same relationships (`:makes`, `:includes`, `:sells`, `:hosts`, `:grants_access_to`) but allows for longer chains, from 3 to 7 hops, thus capturing both direct and indirect connections. The query returns each person's email, the number of distinct festivals reached, and the length of the shortest path to them, ordered by festival count and email.

Query Result:

person	festivals_reached	min_hops
agogerty1e@deliciousdays.com	1	7
aivorym@4shared.com	1	7
alilford1a@ftc.gov	1	7
anewart5@etsy.com	1	7
aoldfieldcherryz@cmu.edu	1	7
bboss19@reuters.com	1	7
bcrucittit@arstechnica.com	1	7
bvowden1@nhs.uk	1	7
castlett7@dyndns.org	1	7
ccapey10@marketwatch.com	1	7
ccarsonv@ft.com	1	7
cchiplen40@hexun.com	1	3
csanzio1c@zimbio.com	1	7
cwasbroughu@blogspot.com	1	7
dbuller15@netlog.com	1	7
dlondon1f@arstechnica.com	1	3
do12@globo.com	1	7
dsansun1d@oakley.com	1	7
eboissierj@tmall.com	1	7
ehinkseng@jigsy.com	1	7
elowisp@sfgate.com	1	7
etwaits70@chron.com	1	3
fburge14@g.co	1	7
fharringtonn@quantcast.com	1	7
ftrillowo@themeforest.net	1	7
gdargavelh@mayoclinic.com	1	7
ggenteryd@geocities.com	1	3
gheymann11@canalblog.com	1	7
girdale1j@bbb.org	1	3
gtiesman4h@prlog.org	1	3
hdu83@vkontakte.ru	1	3
hmanifould16@disqus.com	1	7
ienrietto6@google.ru	1	7
jeglesew@rediff.com	1	7
kdayborne50@senate.gov	1	3
kmary7@bluehost.com	1	7
ksappson1b@gravatar.com	1	7
iellerker7k@w3.org	1	3
maldisc@gnu.org	1	7
mbavidge7b@jalbum.net	1	3
mbergin9@dailymail.co.uk	1	7
mdyshart4l@google.de	1	3
nmacrow31@a8.net	1	3
oskatess@eventbrite.com	1	7
pcressar3@va.gov	1	7
pmcconaghy1i@squarespace.com	1	7
rchilderhouse1h@sogou.com	1	3
rfullmany@blog.com	1	7
rpeacockef@businesswire.com	1	7
sgamwell13@tinyurl.com	1	7
slochhead@cafepress.com	1	7
tcleare1p@army.mil	1	3
tdigwood6h@cisco.com	1	3
tletherbury8@cdc.gov	1	7
vbullickk@ameblo.jp	1	7
wberndtssenb@unblog.fr	1	7
wpochin65@bluehost.com	1	3

Figure 1.14: Query 9bis output

1.3.2.6. Shortest path query

Shortest Path from Person to a "Red Bull" Product:

```
MATCH (pers:person), (prod:product {product_name: 'Red Bull 330ml'})
MATCH p = shortestPath((pers)-[:makes|includes|grants_access_to|hosts|sells*1..8]-(prod))
RETURN pers.person_email_address AS person,
length(p) AS min_hops,
p
```

ORDER BY min_hops ASC, person

LIMIT 5;

Query Description:

The query finds the shortest connection paths between each `:person` node and the `:product` node representing *"Red Bull 330ml"*. It searches for the minimal sequence of relationships among `:makes`, `:includes`, `:grants_access_to`, `:hosts`, and `:sells`, considering paths up to eight hops long. For each person, the query returns their email address, the length of the shortest path (in hops), and the path itself. Results are ordered by path length and person email, displaying only the five shortest connections.

Query Result:

query 10

person	min_hops	p
alliford1a9ffc.gov	2	(:person {person_email_address: alliford1a9ffc.gov, person_full_name: Ardelle Lilford, person_phone_number: 336-239-9940})-[:makes]->(:purchase {purchase_id: P-158996, purchase_payment_method: CARD, purchase_date: 2025-06-10})-[:includes]->(:product:drink {drink_exp_date: 2030-01-04, product_price: 3.5, product_name: Red Bull 330ml})
ccapey10@marketwatch.com	2	(:person {person_email_address: ccapey10@marketwatch.com, person_full_name: Corbet Capey, person_phone_number: 818-548-7486})-[:makes]->(:purchase {purchase_id: P-421812, purchase_payment_method: CARD, purchase_date: 2025-01-06})-[:includes]->(:product:drink {drink_exp_date: 2030-01-04, product_price: 3.5, product_name: Red Bull 330ml})
jwjeaver@rediff.com	2	(:person {person_email_address: jwjeaver@rediff.com, person_full_name: Jerold Eglese, person_phone_number: 199-185-3949})
ksapposort0@gravatar.com	2	(:person {person_email_address: ksapposort0@gravatar.com, person_full_name: Karlis Sapposn, person_phone_number: 558-914-7766})
mbergin99@dailymail.co.uk	2	(:person {person_email_address: mbergin99@dailymail.co.uk, person_full_name: Monica Bergin, person_phone_number: 372-447-6724})

Figure 1.15: Query 10 output

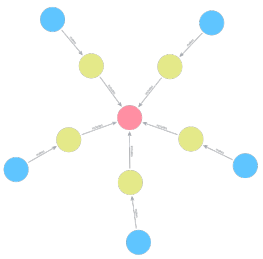

Figure 1.16: Graph representation of the computed path by query 10

2 | BPMN

2.1. Provided process

The Store&Store Company is interested in developing an e-commerce system for their electronic stores, and asked you to provide them with a BPMN diagram depicting the pipeline of interactions with the customer, and the preparation and shipping of an order. The process starts with the customer asking for the available products. The customer service receives the request, and forwards it to the inventory management, which retrieves the list of available products. After receiving the list, the customer prepares its order and sends it to the customer service. The preparation of the order is handled by the inventory management, which checks the availability of each product and, when available, handles its packaging. If any product is not in stock, the customer service sends a notification to the customer, which is required to update the shopping list. Additionally, to allow better management of the warehouse stocks, when a product is running out (i.e., less than 10 items are available), the inventory management places an order to an external supplier before continuing with the order preparation. After all products of the shopping list have been packaged, the customer service notifies the customer that the order is ready for shipping, and then the order is physically sent by the logistics department. Since it is possible that an order may contain defective products, customers are allowed to return items to the customer service. If no item is returned within a week, the order is recorded and the process terminates. Otherwise, the customer service writes a report for the returned items and the process terminates.

2.2. BPMN Diagram

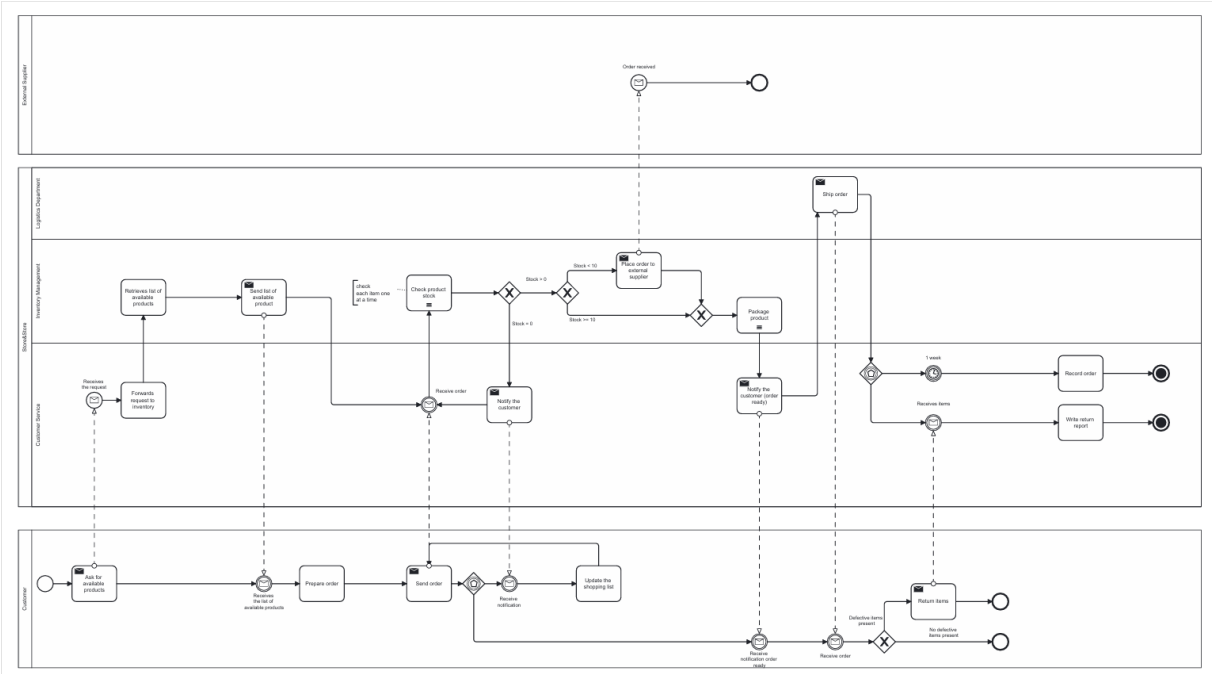

Figure 2.1: BPMN Diagram of the provided text

2.3. BPMN Diagram consideration and assumptions

In this section, we'll try to provide some context and explanations about the choices we made to create the BPMN diagram:

1. We assumed that the product list is a data object implicitly transmitted between activities; therefore, it was not modeled as a physical data artifact but, instead, we decided to introduce a sequential task indicating that, after the order has been received by the customer service, the inventory management starts the process of checking each item one at a time (i.e.: sequentially).

2. A loop occurs if one or more product are sold-out: the customer has to update and resend the order, while the Customer Service wait for it to continue with the process.

3. We explored two possible ways for the XOR gates for product stock. First possibility: checking if there is stock and then if it is needed to ask for a restock. The alternative is to create the flow following a mathematical shrinking-set logic: first we check if the product quantity is 10 or less. If not, the order goes through directly. If not, the product are less than 10, we immediately have to ask for a restock, and then check if we also need to notify the customer (if we have zero products). The first option prioritizes the customer assistance, while the second prioritizes the operations management. We reckon that the first option is more realistic and compliant with the text is sound: in practice, checking availability before deciding on restock better aligns with most process models and user-oriented workflows.

4. We assume that if there are no defective products the customer doesn't try to return its purchase.

5. Two different terminate-all events happen depending on the outcome of the last event gate.

3 | BPMN for Euphoric Events company

We created a mock process coherent with all the logic of our previous work.

3.1. Process: Purchase Merchandise at a Stand

A customer can visit different stands at a festival organised by Euphoric Events to buy limited-edition merchandise; since the merchandise goes sold-out very quickly, the company has implemented a one-sale-per-customer-only policy. A stand is divided in two parts: a front-end desk and a backstage storage section. The front-end is handled by a cashier that communicates directly with the customer, while the storage is managed by a different employee. Each festival has one Main Warehouse Unit (MWU) that stores all ordered merchandise and is responsible for the management of operations such checking product availability and providing products to the stands. An integrated System is implemented to automatically track the merchandise availability of each festival. Additionally, each stand has a personal log, separated from the System, to use to be able to keep track of the items that go sold-out and to be able to notify customers of sold-out immediately. The purchase process starts with a customer visiting a stand and asking for the desired merchandise. The cashier collect the request and immediately checks the stand log: if the product is already ticked as sold-out, the customer is notified, else the request is forwarded to the backstage employee, who checks the product availability in the stand storage. If the requested product is present, the employee confirm the availability to the cashier and brings it to the front end to be sold. The cashier then collect and pack the merchandise and, finally, hands the merchandise to the customer. After having received the product, the customer pays at the stand terminal, which is connected directly to the System which stores the date and the payment method used in the company database. If the requested product is not present in the stand storage, the backstage employee forwards a restock request to the MWU. Whenever the MWU receives a request from one of the stands, a warehouse worker inserts it in the system: if the product is present, the System auto-

matically updates the database by decreasing the product quantity by one, the employee retrieves the product and, at the same time, sends a confirmation to the stand and place it in the pick-up zone. The backstage employee who had initially sent the restock request then has to go pick up the product and bring it to the stand. If no more units of the selected products are present in the MWU, the employee sends a sold-out notification to the stand backstage employee, who forwards it to the cashier. The cashier then update the stand log and notifies the sold-out to the customer.

3.2. BPMN Diagram

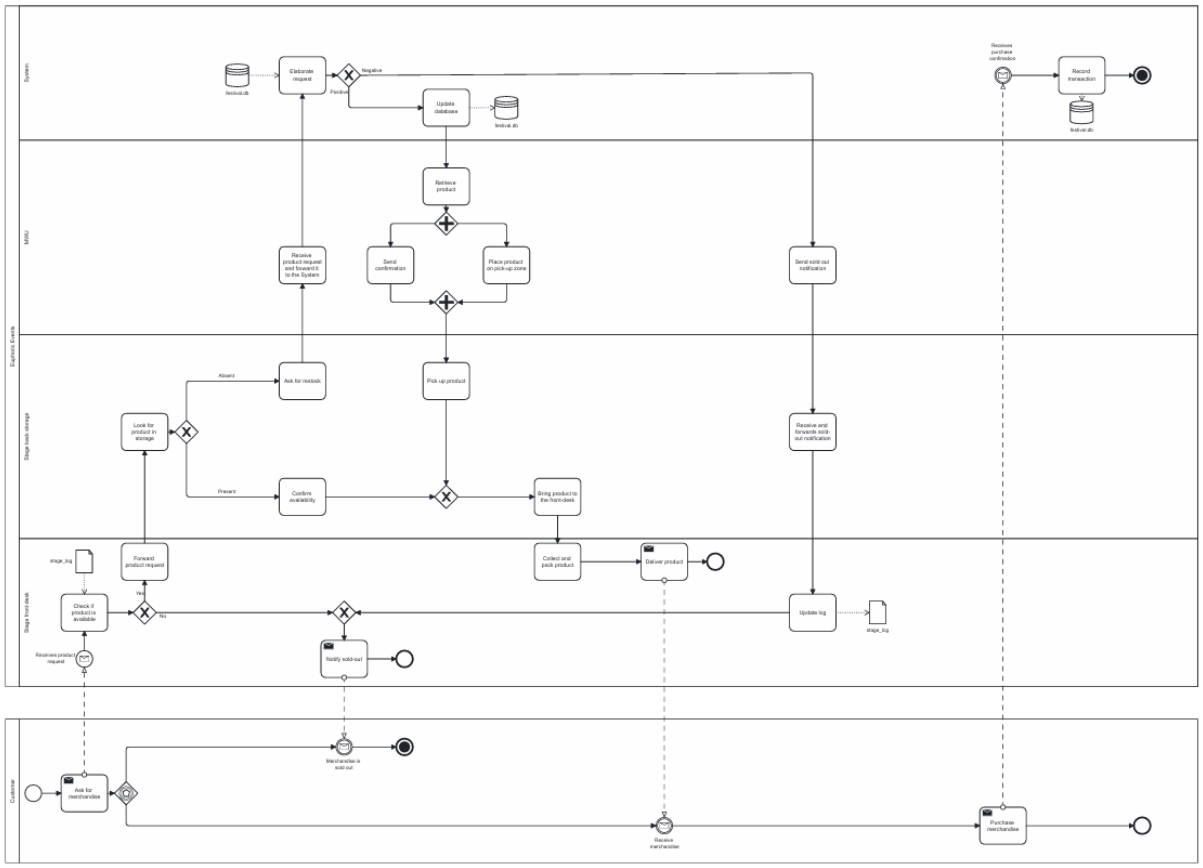

Figure 3.1 BPMN Diagram for Euphoric Events company

3.3. BPMN Diagram assumptions

1. The model was created trying to be as realistic as possible. There are many other, more efficient and smoother processes that could be implemented.
2. The stand_log is a local tool to aid cashiers, which may be an .xlsx file accessible by the stand or even a paper log.
3. The original RDBMS from our previous work is represented here by the System, which has its own lane and is the only entity allowed to access and modify the company database. The "festival.db" is represented as an artifact within the System lane and is only accessible by the System itself.
4. Each Stand and the MWU has a terminal to communicate with the System.
5. The customer pays once he/she receives the desired item.